

SYBEX Supplement

Mastering Windows 2000 Programming with Visual Basic C++

by Ben Ezzell

Screen reproductions produced with Collage Complete.
Collage Complete is a trademark of Inner Media Inc.

SYBEX, Network Press, and the Network Press logo are registered trademarks of SYBEX Inc.
Mastering, Expert Guide, Developer's Handbook, and No experience required. are trademarks of SYBEX Inc.

TRADEMARKS: SYBEX has attempted throughout this book to distinguish proprietary trademarks from descriptive terms by following the capitalization style used by the manufacturer.

Netscape Communications, the Netscape Communications logo, Netscape, and Netscape Navigator are trademarks of Netscape Communications Corporation.

Microsoft® Internet Explorer ©1996 Microsoft Corporation. All rights reserved. Microsoft, the Microsoft Internet Explorer logo, Windows, Windows NT, and the Windows logo are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

The author and publisher have made their best efforts to prepare this book, and the content is based upon final release software whenever possible. Portions of the manuscript may be based upon pre-release versions supplied by software manufacturer(s). The author and the publisher make no representation or warranties of any kind with regard to the completeness or accuracy of the contents herein and accept no liability of any kind including but not limited to performance, merchantability, fitness for any particular purpose, or any losses or damages of any kind caused or alleged to be caused directly or indirectly from this book.

Photographs and illustrations used in this book have been downloaded from publicly accessible file archives and are used in this book for news reportage purposes only to demonstrate the variety of graphics resources available via electronic access. Text and images available over the Internet may be subject to copyright and other rights owned by third parties. Online availability of text and images does not imply that they may be reused without the permission of rights holders, although the Copyright Act does permit certain unauthorized reuse as fair use under 17 U.S.C. Section 107.

Copyright ©2000 SYBEX Inc., 1151 Marina Village Parkway, Alameda, CA 94501. World rights reserved. No part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, including but not limited to photocopy, photograph, magnetic or other record, without the prior agreement and written permission of the publisher.

S U P P L E M E N T

O N E

S1

Message Handlers and the Microsoft Foundation Classes

- Message-handling formats
- WM_PAINT message processing
- Windows font metrics and measurements
- Windowing text output
- Window resizing

In the introductory chapters to this book (specifically Chapter 2) we discussed the *WinHello* and *WinHello2* programs, which displayed a simple message in the center of the application window, and then forgot about them. This is all that the demo programs were intended to do; they were provided with only minimal capabilities because the object was simply to demonstrate how a Windows program should be initialized and how to use a simple message handling loop.

In this chapter, we will use another, slightly less simple, program to demonstrate several elements used in Windows applications. The *PainText* demo demonstrates writing a display larger than the application window, along with provisions for scrolling the display within the window. Also, unlike the *WinHello* demo, in which only one message is important, the *PainText* example responds to several event messages—a much more realistic eventuality. Another program discussed in this chapter is *PainText2*. This program handles the same tasks as *PainText*, but it uses the MFC classes.

Both programs also demonstrate an aspect of display handling that is necessary for Windows applications: re-creating the screen, in part or entirely, as required.

PainText versus PainText2: Conventional versus MFC Message Handling



For many programmers who began creating applications under DOS (or another early operating system, such as Unix or CP/M), the change to Windows' event-message programming has required an adjustment in attitude and in their approach to programming. Creating a program as responses to event messages rather than a direct flow of actions is a very different milieu.

By now, after several generations and versions of both Windows and OS/2, event-driven programming is not only acceptable and convenient, but often it is the method of choice for triggering an action or activity, even when doing so requires defining and generating custom messages.

In a conventional Windows application, the message responses are normally handled in the *WndProc* procedure as a *switch/case* statement, where the various case statements may call subprocedures or may contain the code for the immediate response.

In an MFC-based application, however, the conventional `WndProc` procedure has been replaced by a message-map handler which directs the event messages to specific class methods that provide the responses. At the same time, the message mapping often also interprets the conventional message parameters in a more convenient format.

In the examples discussed in this chapter, using MFC for the *PainText2* demo introduces two principal changes from the *PainText* program:

- Instead of a message-handling loop, `CALLBACK` functions cause message events to connect directly to the event-response functions. There is still a message loop, but using MFC, it is effectively hidden from view.
- Rather than providing functions to handle scrolling as in *PainText*, the *PainText2* program uses the `CScrollView` class, allowing the MFC library to handle the scrolling for us. Functionally, the same tasks are performed but the handling is greatly simplified.

As a brief example, the switch/message handler for the *PainText* program (conventional version) is shown in Table S1.1, along with the equivalent MFC-class methods from the *PainText2* version and the `CScrollView` equivalents.

TABLE S1.1: Conventional versus MFC Message Handling

Conventional	MFC Equivalent	CScrollView Equivalent
<code>switch(msg)</code>		
<code>{</code>		
<code>case WM_CREATE: ...</code>	<code>OnCreate ...</code>	<code>OnCreate ... / OnUpdate ...</code>
<code>case WM_SIZE: ...</code>	<code>OnSize ...</code>	<i>Handled by CScrollView</i>
<code>case WM_PAINT: ...</code>	<code>OnDraw ...</code>	<code>OnDraw ...</code>
<code>case WM_VSCROLL: ...</code>	<code>OnVScroll ...</code>	<i>Handled by CScrollView</i>
<code>case WM_HSCROLL: ...</code>	<code>OnHScroll ...</code>	<i>Handled by CScrollView</i>
<code>case WM_DESTROY: ...</code>	<i>(Uses default handler)</i>	<i>Handled by CScrollView</i>
<code>}</code>		

While it is perfectly practical—and sometimes necessary—to incorporate an old-style `switch/case` statement in an MFC `OnCommand` function, the newer format is more convenient. Furthermore, by using the `CScrollView` class, most of the messages that the conventional application needed to handle are now handled automatically without requiring any provisions in the application.

The following sections discuss the conventional message-handling functions, referred to in the *PainText* application, while the MFC equivalents, from the *PainText2* version, are covered primarily where they differ from conventional message-handling functions or require special responses. At the end of the chapter, we will focus on the *PainText2* version and how MFC can simplify the same process by automating most of the operations required by the *PainText* demo.

NOTE

The *PainText* and *PainText2* demos are included on the CD in the Chapter 3 folder.

Screen-Recovery Operations

Under DOS, once the screen is written, an application is pretty well free to forget about it and proceed with something else. Under Windows, however, even though an application is limited to its own client window, the display created is not inviolate, and the application must be prepared to re-create the display as required. And because the application does not “own” the display, it must be prepared for its display to be invalidated—by another window overlaying its display, by its display being resized or shrunk to an icon and restored, or by its application window simply being moved on the screen. The application itself may invalidate its display by overwriting some portion with a pull-down menu or a pop-up dialog box.

For a text-based display, pop-up dialog boxes and pull-down menus can provide their own screen recovery by saving a memory copy of the existing display and, when dismissed, can erase themselves by restoring the original display from memory. In text modes, this is relatively simple because less than 4KB are necessary to save an entire screen (80×25×2 bytes—character and attribute—per cell).

For a graphics display, however, a similar operation would require nearly 300KB, assuming a screen 640×480 with 16 colors. Of course, for SVGA and True

Color displays, memory requirements increase accordingly. Granted, data compression could reduce these requirements to some degree, but until super-fast terabyte memories become common, the saved image approach is not likely to be considered practical under general circumstances. Instead, Windows applications are expected to be able to re-create the screen as required.

NOTE

There are circumstances in which Windows does save overwritten display areas, such as when the cursor overwrites the display or when an icon is dragged across a client area. Under these limited circumstances, no screen update is required. However, in all other cases, Windows notifies the application whose screen displays have been invalidated when it is appropriate to re-create the client window while Windows handles restoration of the application's frame.

The WM_PAINT Message

The WM_PAINT message is posted to an application as notification that the current screen display is invalid, requiring restoration. Thus, a WM_PAINT message is issued, notifying the application that it's time to repaint its display, under these conditions:

- When an application window has been hidden, partially or entirely
- When an application window has been resized (assuming the CS_HREDRAW and CS_VREDRAW flags were set in the style specification)
- When ScrollWindow is called to scroll the client area, horizontally or vertically

At the same time, there are circumstances under which the application may wish to issue its own WM_PAINT message. For example, during initialization, most applications call the UpdateWindow function, which instructs Windows to issue a WM_PAINT message addressed to the application's client window. Then, after the message loop begins processing, the WM_PAINT message is picked up and forwarded to the WndProc function, and finally, the initial window display is painted.

In other circumstances, the application may choose to use the InvalidateRect or InvalidateRgn functions, both of which explicitly generate WM_PAINT messages, along with information specifying the area requiring repainting. Applications written using MFC may simply call the Invalidate function for the same result.

At first, this may appear to be a rather roundabout means of accomplishing what, in other circumstances, would be a fairly straightforward task. After all, instructing Windows to send a message back to the application to request a repaint is a bit like riding ‘round Robin Hood’s barn.

NOTE

“Robin Hood’s barn” refers to Sherwood Forest and is a common expression for a task carried out in the hardest possible manner.

The reasons, however, are far more than philosophical. For multiple applications to share a computer, as is the case under Windows, they must operate in a fashion permitting others to have access to the system resources. To accomplish this, applications are required to break their operations into a series of subtasks and, instead of initiating these tasks directly, to place *requests* (that is, messages) in a queue. Control of the system is passed back to Windows as each task is completed, and if necessary, Windows can then pass control to another application. The result is flexible time sharing, with Windows offering each application time and resources according to its needs.

The important item to remember is that applications must be prepared to recreate their display space *at any time*. They must be ready to write the screen on demand and to rewrite the screen on demand.

The PAINTSTRUCT Structure

Hand in glove with the WM_PAINT message is the PAINTSTRUCT information structure. A separate PAINTSTRUCT record is maintained by Windows for each application, with the structure defined as:

```
typedef struct tagPAINTSTRUCT
{
    HDC    hdc;
    BOOL   fErase;
    RECT   rcPaint;
    BOOL   fRestore;
    BOOL   fIncUpdate;
    BYTE   rgbReserved[32];
} PAINTSTRUCT, *PPAINTSTRUCT, *NPPAINTSTRUCT,
  *LPPAINTSTRUCT;
```

The first three fields in `PAINTSTRUCT`—`hdc`, `fErase`, and `rcPaint`—are commonly used by applications. The latter three fields are used internally by Windows 98.

NOTE

If the `rgbReserved` field (the sixth field) is accessed directly when converting from Windows 3.x to Windows 98, be aware that the size of this field has changed from 16 to 32 bytes.

The first field, `hdc`, is simply a handle to the application's device context. Rather than accessing the `hdc` field from the `PAINTSTRUCT` field, however, applications should continue to depend on the value returned by the `BeginPaint` or `GetDC` functions called before any screen-update operations commence.

The second field, `fErase`, is a flag value with, confusingly, `FALSE` instructing Windows to erase the background of an invalidated rectangle, and `TRUE` indicating that the background has already been erased.

The third field, `rcPaint`, consists of a `RECT` structure defined as:

```
typedef struct tagRECT
{
    LONG    left;
    LONG    top;
    LONG    right;
    LONG    bottom;
} RECT, *PRECT, NEAR *NPRECT, FAR *LPRECT;
```

The `rcPaint` field is used to keep track of the invalidated region within the application's client window. The four values in the `rcPaint` field define the sides of the smallest rectangle enclosing all invalidated areas.

When an application is finished responding to a `WM_PAINT` message (by calling `EndPaint`), the `rcPaint` field is reset, validating the entire client window. Subsequently, when some portion of the client window is overwritten by another application, pull-down menu, or pop-up dialog box, a new invalidated region is calculated. Likewise, when any application is moved, closed, or resized, Windows checks for other applications affected by these changes, resetting the invalidated areas as required and, as appropriate, posting `WM_PAINT` messages to instruct applications to restore their display areas.

NOTE

When you are using MFC and responding to the `OnDraw` method call—the MFC equivalent to a `WM_PAINT` message—the device context is supplied as a pointer, `pDC`, to the active device context. Also, under MFC, the `BeginPaint/ EndPaint` functions are not required. Applications using MFC do not have direct access to the `PAINTSTRUCT` structure.

The purpose of the invalidated rectangle is twofold:

- Because paint operations are restricted to the area specified, an application overlapped by another application's window—or even by another of its own display elements—does not overwrite the higher-level display while restoring its own display area.
- This method restricts the area that requires repainting to the minimum actually necessary. While text-based displays can afford less-than-optimum screen updates without being visually apparent, graphics displays, requiring more processing, lack this luxury. To present a smooth, visually seamless display, they must use the optimum approach of executing the update in the shortest possible time, which also means within the smallest possible area.

Applications may also need to set their own update areas. This task is accomplished by calling the `InvalidateRect` function:

```
InvalidateRect( hwnd, NULL, TRUE );
```

The first parameter, `hwnd`, is the window handle. The second parameter specifies the region to be invalidated. Specifying the region as `NULL`, as in this example, is the equivalent of specifying the entire client area. Alternatively, you could use an `HRGN` argument to pass a handle to a data structure containing the precise region coordinates. If the third argument is passed as `TRUE`, it erases the background for the set region; if it's `FALSE`, it leaves the current background unchanged.

Painting Operations

While Windows is responsible for issuing the majority of the `WM_PAINT` messages, the application is responsible for responding to these messages and for creating

or re-creating the application display as necessary. However, before the application can draw anything—even a single pixel—the application must begin by obtaining the device-context handle (commonly abbreviated `hdc`).

In the *WinHello* demo, the device-context handle is returned by calling the `BeginPaint` function as:

```
hdc = BeginPaint( hwnd, &ps );
```

In this fashion, the application has not only obtained a handle to the device context but also, at the same time, retrieved the `PAINTSTRUCT` record (`ps`) by passing the address of a local variable of the appropriate type. The form shown is commonly used in response to `WM_PAINT` messages and is always matched, when the current operations are finished, with a corresponding `EndPaint` function call:

```
EndPaint( hwnd, &ps );
```

If you are using MFC, however, the painting operations are encapsulated in the `OnDraw` method instead of the `BeginPaint` and `EndPaint` instructions. Within the `OnDraw` method, the device context is supplied as an argument, but painting operations proceed in the same fashion as in response to the `WM_PAINT` message (see the parallel examples in later chapters).

In other circumstances, a second method of accessing the device context is:

```
hdc = GetDC( hwnd );
```

Or, using MFC, the `CWnd::GetDC` method is invoked to return a pointer to the device context:

```
CDC* pDC;  
pDC = GetDC();
```

The `GetDC` function is commonly used in any situation where immediate client window operations cannot wait to respond to a `WM_PAINT` message. For example, a Clock program, responding to a timer event, needs to update its image immediately and cannot simply wait for a `WM_PAINT` message to appear in the queue.

NOTE

The `GetDC` function is not restricted to paint operations; it is also used when an application requires information from a device context. For an example, refer to the font and text metrics example in Supplement 14.

Like the `BeginPaint` function, the `GetDC` function has its own closing statement:

```
ReleaseDC( hwnd, hdc );
```

Using MFC, instead of requiring a window handle, the `CWnd::ReleaseDC` method is called:

```
ReleaseDC( pDC );
```

WARNING

`BeginPaint` must always end with `EndPaint`. `GetDC` is always closed with a `ReleaseDC` function call. Mixing these functions incorrectly will not produce a compiler error but will have serious, or possibly fatal, effects on an application's execution.

Okay, why two formats? Because each format has a different purpose, and each operates in a different fashion.

The `BeginPaint/EndPaint` process, as mentioned previously, returns and resets the invalidate region data, but it also restricts drawing operations to the region specified.

The `GetDC/ReleaseDC` process returns a clipping rectangle, which is equal to the entire client window, imposing no restrictions on drawing operations (aside from the inherent limitation to the application's window). At the same time, while `EndPaint` resets the invalidated region, `ReleaseDC` does not change existing settings and, therefore, does not clear information that might be needed later to restore an invalidated area.

NOTE

While `GetDC` permits drawing operations over the entire client window area, Windows itself prevents these operations from overwriting an overlying application's window area, and restricts screen operations to the visible or physical portion of the display.

The `GetDC` and `ReleaseDC` functions are frequently used when only information about a device context—whether the display, a printer, or some other device—is required. You'll see this use demonstrated in the next section.

Controlling Graphics Text Displays

Within the Windows environment, four primary factors govern how text is drawn:

Position The row/column absolute screen positions used in a text environment are replaced, in Windows, with window-relative pixel coordinates. Positioning must also take into account *font metrics* (text sizing), alignment options, and *scroll positioning* (vertical and horizontal), as well as variable window sizing.

Text size and alignment In DOS text mode, characters are a fixed size and positioned automatically by the cursor position or by explicit row/column directions. In Windows, as with other graphics environments, text sizes, styles, and fonts can be mixed, and with the exception of a few fixed-width fonts, individual characters vary in size. Regardless of font, characters and/or strings are positioned by pixel coordinates, not by row and column.

Scrolling DOS text mode displays, conventionally, are limited to unidirectional vertical scrolling. When horizontal scrolling is permitted, movement is based on character columns. In Windows, both vertical and horizontal scrolling can be adjusted in single-pixel steps or in any other increment the developer desires. Text displays must take into account offsets from origin points, providing their own vertical (line) calculations. Fortunately, in most cases, horizontal positioning can simply be handled as an offset, without complex calculations.

Window limits DOS text displays can depend on autowrap to prevent too-long strings from extending beyond the physical display. In Windows, the virtual and physical displays do not share the same limits; therefore, applications must provide their own length calculations and line breaks. In some applications, text is sized to a phantom, virtual screen's limits, requiring scrolling to view various portions of the virtual window. In other cases, applications may reformat text to accommodate changes in window size.

In the Windows environment, these four elements are not entirely separate considerations. Instead, they tend to be interrelated or even synergistic in their effects. And while these relationships present their own problems, Windows shields you from many of the other problems that otherwise would be part and parcel of the process of sharing a variable-sized display in a multiple-application environment.

There are advantages as well. For one, because operations are always relative to the window, applications can be moved around the screen without the application requiring special provisions for repositioning. For another, the application itself does not need to recognize the hardware, screen size, and other display constraints and adjust its behavior

Continued on next page

accordingly. Also, though less commonly a consideration, Windows itself provides a variety of display fonts as well as offering accessibility to additional third-party fonts.

Of course, the real point is simply that Windows applications must take a different approach to writing any type of screen display than a similar application operating in the DOS environment.

Windows Font Metrics and Measurements

The *WinHello* demo used the simplest possible text output, employing the `DrawText` function to write a single line centered in the application's client window. However, while the demo is suitable for brief text in a very simple context, most applications will require displays with more than one line of text and/or more sophisticated positioning.

For displays with multiple lines of text, two pieces of data are essential: vertical line spacing and horizontal line length (assuming a horizontal orientation). But neither of these characteristics is fixed; they both depend on font selection and, without knowing the relevant text metrics, cannot be arbitrarily assumed.

Also, even for system fonts, you cannot assume that font characteristics will be the same for all systems because, during installation, Windows matches fonts to the video-display capabilities. At the same time, video board manufacturers and third parties design and distribute their own system fonts as well as specialty fonts.

Thus, regardless of font selection, applications must treat the font metrics as variables and request the current font information through the `GetTextMetrics` function:

```
TEXTMETRICS    tm;

hdc = GetDC( hwnd );
GetTextMetrics( hdc, &tm );
ReleaseDC( hwnd, hdc );
```

This also provides an example of using the `GetDC` function in place of the `BeginPaint` function. Since no screen-painting operations are executed, there's no need for invalidated region information or for `PAINTSTRUCT` data. Thus, for a simple information retrieval, only the `GetDC` operation is necessary.

The `TEXTMETRIC` structure is defined as:

```
typedef struct tagTEXTMETRIC
{
    LONG    tmHeight;
    LONG    tmAscent;           LONG    tmDescent;
    LONG    tmInternalLeading;   LONG    tmExternalLeading;
    LONG    tmAveCharWidth;    LONG    tmMaxCharWidth;
    LONG    tmWeight;          LONG    tmOverhang;
    LONG    tmDigitizedAspectX; LONG    tmDigitizedAspectY;
    BYTE    tmFirstChar;       BYTE    tmLastChar;
    BYTE    tmDefaultChar;     BYTE    tmBreakChar;
    BYTE    tmItalic;          BYTE    tmUnderlined;
    BYTE    tmStruckOut;       BYTE    tmPitchAndFamily;
    BYTE    tmCharSet;         } TEXTMETRIC;
```

Of these twenty fields, the seven that control text spacing are illustrated in Figure S1.1.

FIGURE S1.1:

Windows font metrics

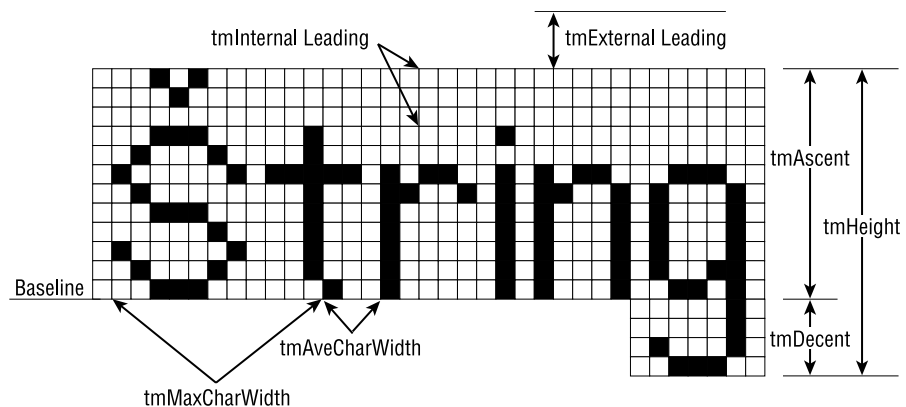


Figure S1.1 shows the following `TEXTMETRIC` fields:

tmInternalLeading Provides space for accent marks above characters as illustrated by the S-umlaut combination shown in Figure S1.1. Although accents are not commonly used in English, many other languages depend heavily on interlinear accent marks. In some cases, international character

sets provide for these; in other cases, such as Thai, algorithms are employed to correctly combine characters, tone accents, and vowel marks. In all cases, this spacing should be the absolute minimum between lines.

tmExternalLeading Provides the font designer's suggested interlinear spacing. Although it's optional, white space between text lines increases the readability of the display.

tmAscent Specifies the height of an uppercase character, including the **tmInternalLeading** space.

tmDescent Provides space for character descenders, as in the characters *g*, *j*, *p*, *q*, and *y*.

tmHeight Identifies the overall height of the font, including the **tmAscent** and **tmDescent** values, but not the **tmExternalLeading** value.

In the *PainText* demo, the vertical line spacing used, *cyChr*, is calculated as:

```
cyChr = tm.tmHeight + tm.tmExternalLeading;
```

The length of a text string is not calculated as simply, however. **TEXTMETRIC** supplies two values for character widths:

tmAveCharWidth Specifies width calculated as the weighted average of the lowercase character widths.

tmMaxCharWidth Specifies width as the single widest character in the font, usually either the *W* or the *M* character.

If calculating the width of a character string is critical, you can approximate a third value for the average width of the uppercase letters for most fonts as 150 percent of the **tmAveCharWidth** value.

Text Sizing

Since font information will remain unchanged during program execution (unless a new font is selected, of course), the simplest method of calculating text spacing is to retrieve the text metrics information when the application is initiated; that is, in response to the **WM_CREATE** message, the first message **WndProc** receives.

Text sizing is accomplished in the *PainText* demo as:

```
case WM_CREATE:
    hdc = GetDC( hwnd );
    GetTextMetrics( hdc, &tm );
    ReleaseDC( hwnd, hdc );
    cxChr = tm.tmAveCharWidth;
    cxCap = (int)( cxChr * 3 / 2 );
    cyChr = tm.tmHeight + tm.tmExternalLeading;
    break;
```

This provides three basic values for positioning text using the current font: the average lowercase character width, the average uppercase character width, and the vertical line spacing (although only the *cxChr* and *cyChr* values are needed in the *PainText* demo).

In the MFC version of the *PainText* demo, *PainText2*, the *Create* method might appear to be the appropriate location for the corresponding code. However, attempting to call the *GetDC* function from the *Create* method will fail because the appropriate *CWnd* class has not yet been initialized. Instead, you could use the *OnCreate* method, which is the equivalent to the *WM_CREATE* message response. Alternatively, as in the *PainText2* demo, the *OnShowWindow* method serves nicely:

```
void CPainText2View::OnShowWindow(BOOL bShow, UINT nStatus)
{
    CScrollView::OnShowWindow(bShow, nStatus);

    if( bShow )
    {
        TEXTMETRIC  tm;
        CDC          *pDC;

        pDC = GetDC();
        pDC->GetTextMetrics( &tm );
        m_cyChr = tm.tmHeight + tm.tmExternalLeading;
        m_cxChr = tm.tmMaxCharWidth;
    }
}
```

Here, the *bShow* argument is tested to determine if the window is being shown or hidden, although the fact that the *OnShowWindow* method is being called at all should be sufficient assurance that we do have a window and, therefore, can call the *GetDC* method to retrieve a device context.

Notice, however, that unlike in the *PainText* demo, there is no instruction to call the `ReleaseDC` method because this is done automatically when the `CDC` class goes out of context. Also, we will use the retrieved information in a slightly different way, as you will see later in the chapter.

Window Coordinates and Limits

In DOS text mode, the row/column coordinate system is based on an origin point at the upper-left corner of the screen, beginning at 1,1. Under Windows, the default coordinate system used is nominally the same, with three provisions:

- The coordinates are relative to the window.
- The coordinates are in pixel, not row/column, units.
- The origin point is numbered 0,0, not 1,1.

NOTE

Windows provides several mapping modes, each employing different scalar units and different coordinate origins. For text displays, only the default text mapping mode is required.

Because Windows applications do not write directly to the screen—only indirectly through Windows API functions—applications do not need to know where their client windows lie in relationship to the physical screen. Instead, Windows applications simply write to their own virtual screens, using virtual (window-relative) coordinates, and they leave the mapping from the virtual to the physical to Windows.

When relevant, applications may limit operations to the present width and height of their client window. Alternatively, they may write to an assumed screen of optimum width and/or height (even if this is larger than the active window dimensions) and rely on scrollbar operations to position the *viewport* (the visible window) over the virtual display, as demonstrated by the *PainText* demo.

Outputting Text to a Window

After retrieving the `TEXTMETRIC` information and deriving the height and width information for the system font, the next obvious step is to write text to the client window (the application's screen display). In the *WinHello* demo, the `DrawText` function is sufficient. However, in *PainText*, a more sophisticated display is provided by using the `TextOut` function.

Like the *WinHello* demo, the *PainText* demo follows the standard response pattern of painting the application screen in response to a `WM_PAINT` message. However, because the text displayed will be larger than the application window, the client window also has vertical and horizontal scrollbars to position the viewport over a larger virtual window.

The text metrics information has already been retrieved when the `WM_CREATE` message is received, but at this time, there is additional information that the application requires. Part of this information can be derived from the `ps` paint structure.

```
case WM_PAINT:
    hdc = BeginPaint( hwnd, &ps );
    nFirst = max( 0, cyPos + ps.rcPaint.top / cyChr );
    nLast = min( NUM_LINES, cyPos + ps.rcPaint.bottom / cyChr );
```

NOTE

`cyPos` is calculated in lines of text, not in pixels. In like fashion, the `cxPos` (horizontal) is measured in average character widths.

The `cyPos` variable contains the present vertical scrollbar settings (initially set to zero); `cyChr` is the vertical line spacing. The `ps.rcPaint.top` and `ps.rcPaint.bottom` arguments identify the top and bottom, respectively, of the window's invalidated rectangle area (these values are relative to the client window, of course).

In the MFC version (*PainText2*), we use a `CRect` instance to retrieve the client window coordinates (window size) and to calculate the beginning and ending lines for our display, because the `ps` structure is not available in the `OnDraw` method.

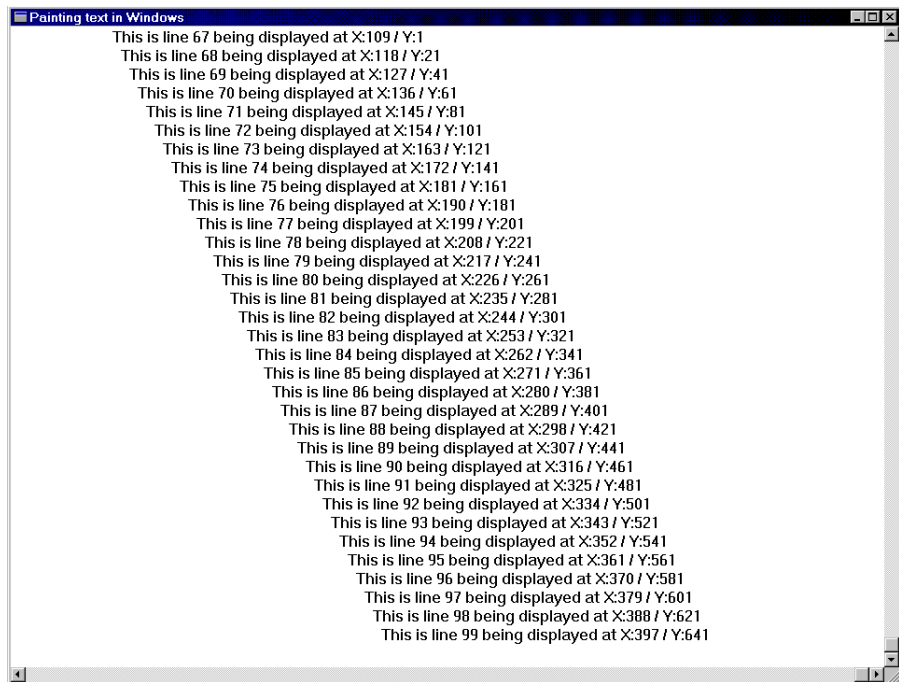
```
GetClientRect( cRect );
nFirst = max( 0, m_cyPos );
nLast = min( NUM_LINES, m_cyPos + cRect.Height() / m_cyChr );
```

Another difference is the variable designation `m_cyPos` instead of simply `cyPos`. `m_cyPos` indicates a member variable; that is, a variable belonging to the class rather than to a local function. Member variables are globally available to all class methods. Local variables are not available outside of the method where they are declared.

Given this information, the calculated `nFirst` and `nLast` values will provide line numbers identifying text that needs to be repainted, as shown in Figure S1.2.

FIGURE S1.2:

Displaying multiple text lines



The two macros `min` and `max` are employed simply as a safeguard against errors in calculation. They are used to ensure that `nFirst` is never less than 0 and that `nLast` cannot be greater than `NUM_LINES`—the maximum number of lines that will be written.

Text Alignment

After the beginning and ending points have been determined, the next step is to set the appropriate text alignment before writing anything to the screen. Under DOS, text alignment is fixed, but in almost any graphics context, a choice of alignments is permitted. In this case, the `TA_LEFT` and `TA_TOP` settings provide that the text string will be written with the top and left extents aligned with the output coordinates:

```
SetTextAlign( hdc, TA_LEFT | TA_TOP );
for( i = nFirst; i <= nLast; i++ )
{
    x = 1 + cxChr * ( i - cxPos );
    y = 1 + cyChr * ( i - cyPos );
```

Here, the MFC version is virtually identical. The only real difference is in how the `SetTextAlign` function is called.

```
pDC->SetTextAlign( TA_LEFT | TA_TOP );
for( i = nFirst; i <= nLast; i++ )
{
    x = 1 + m_cxChr * ( i - m_cxPos );
    y = 1 + m_cyChr * ( i - m_cyPos );
```

New x-axis and y-axis screen positions (within the client window) are calculated for each line written, together with an x-axis offset to indent successive lines. Normally, the x-axis position used would be one character width (inset from the client window frame). In this case, a progressive offset is used to produce a display that is wider than any normal display terminal can handle and, thus, demonstrates horizontal scrolling.

Also, remember that the positions calculated take into account the scrollbar offsets. Therefore, either or both values may be negative integers, indicating that the current line begins outside the active client window. This is not an error—screen-painting operations may originate at coordinates outside the window, either in the negative or positive directions. When this happens, or when drawing operations extend outside the client window, Windows simply truncates the actual painting operation to the visible region, without requiring the application to make elaborate and complex accommodations.

The alternative, attempting to calculate where a string should be truncated in order to fit the active window and present the appropriate alignment, is not only cumbersome but, in practical terms, effectively impossible when a variable-width font is used.

On the other hand, asking the application to begin by writing several hundred lines of text above the visible screen (and hundreds more below), simply to include the visible portion of the display, would be both slow and unnecessarily cumbersome. Ergo, the simplest approach is to allow the application to execute its operations in a virtual space that is as large as necessary horizontally, with Windows providing clipping, but at the same time, provide reasonable beginning and ending points vertically. This approach is demonstrated in the *PainText* demo.

Formatted Text Output

In conventional programs, formatted text output is provided directly using the `printf` function (or an equivalent):

```
gotoxy( x, y );  
printf( "This is line %d being displayed"  
        " at X:%d / Y:%d", i, x, y );
```

In Windows, a somewhat different approach is required:

```
TextOut( hdc, x, y, szBuffer,  
         wsprintf( szBuffer,  
                   "This is line %d being displayed"  
                   " at X:%d / Y:%d", i, x, y ) );
```

The `TextOut` function expects five parameters:

- The device-context handle (`hdc`)
- Two screen coordinates (`x` and `y`)
- A long pointer (`LPSTR`) to an ASCIIZ string to be written

NOTE

The term *ASCIIZ* is shorthand for a null-terminated ASCII string.

- The length of the string (in characters)

These parameters could be provided in a series of separate steps. For example, you could use the `sprintf` function to format the string to a buffer (an array of `char`), and then pass the buffer, together with its length, to the `TextOut` function. However, because the `wsprintf` function returns the string length directly while writing the text to a buffer, multiple separate instructions can be reduced to a single longer instruction. (Arguments are always evaluated from left to right; that is, in the same order listed.)

In the MFC version, instead of using a `char` array, a `CString` object is used to create the string. We do not need to supply a string length as a separate argument because the length is included in the `CString` object.

```
csText.Format( "This is line %d being displayed at"  
               " X:%d / Y:%d", i, x, y );  
pDC->TextOut( x, y, csText );
```

Notice that in all three cases—whether the `printf`, `wsprintf`, or `CString::Format` function is used—the arguments and format instructions are the same.

To sum up, it's a case of sixes and half-dozens, with little to choose among the three except personal preferences.

Scrollbar Operations

Scrollbars are a popular control feature normally associated with screen displays for adjusting the horizontal and vertical positioning (although scrollbars are seeing increasing use for other scalar adjustments). Perhaps the only drawback to scrollbars is that they frequently cannot be used without a mouse (many applications implement the arrow and page keys as alternative controls). However, since few (if any) Windows users lack a mouse, there is certainly no reason not to use scrollbars and every reason, including familiarity and programming convenience, to employ them.

Of course, there is always at least one fly in the ointment. For scrollbars, the fly is that the scrollbar operations are not automatic; applications require a few provisions before they can respond to scrollbar messages.

Scrollbar Messages

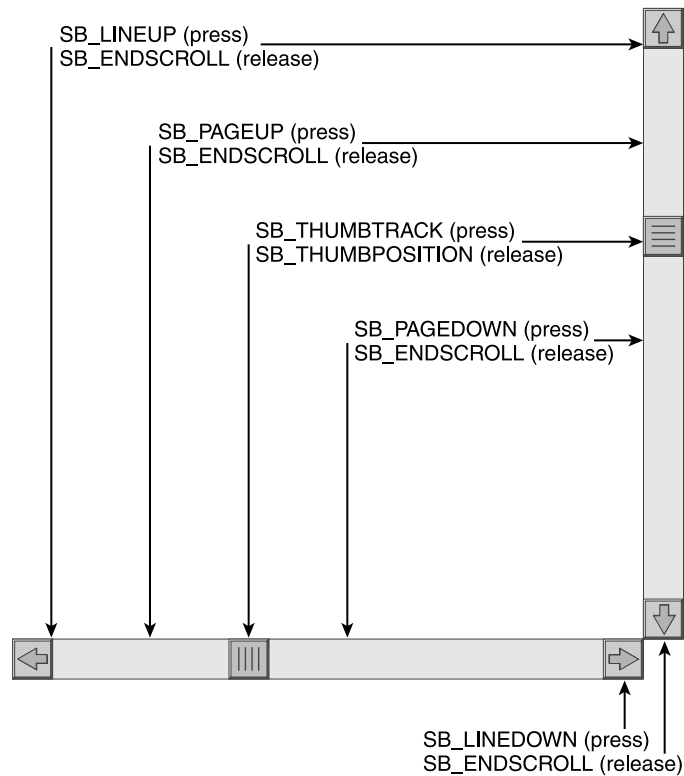
Figure S1.3 illustrates two scrollbars (vertical and horizontal), with labels showing the Windows messages posted when you click each scrollbar region with the mouse or release the mouse.

Each scrollbar has five active regions (unless it is created in too small a size): the two end arrows (endpads), the thumbpad, and the scrollbar body on each side of the thumbpad. When the mouse is clicked or released on any of these areas, each generates a different set of event messages, as shown in Figure S1.3.

Both scrollbars return `SB_LINEUP` messages when the mouse is clicked (button down) on the top or left endpad; they return `SB_LINEDOWN` messages when the mouse is clicked on the bottom or right endpad. Alternatively, if the mouse button is held down on any of the endpads, a continuous series of `SB_LINEUP` or `SB_LINEDOWN` messages is generated, providing continuous scrolling in the appropriate direction.

FIGURE S1.3:

Scrollbar messages



The body of the scrollbar—the area between either endpad and the thumbpad—is also an active control. If the mouse hit is above or to left of the thumbpad, an `SB_PAGEUP` message is generated. When the mouse hit is to the right of or below the thumbpad, an `SB_PAGEDOWN` message is generated. When the mouse button is released anywhere except on the thumbpad, an `SB_ENDSCROLL` message is returned.

The thumbpad itself generates a different type of message. It returns a series of `SB_THUMBTRACK` messages as long as the mouse button is down. When the mouse button is released, the thumbpad returns a single `SB_THUMBPOSITION` message, as long as the mouse cursor is still on the scrollbar. If the mouse cursor has moved off the scrollbar, no release message is posted.

Also, even if the `SB_THUMBTRACK` messages are ignored (by the application), as long as the mouse button is held down and the mouse remains on the scrollbar, Windows generates a thumbpad outline following the mouse position.

Scrollbar Ranges and Thumbpad Positions

In order for scrollbars to report and function correctly, each scrollbar must have both the range and thumbpad positions assigned.

For a standard Windows application, the `SetScrollbarRange` function is called with the application window's handle, an integer constant identifying the scrollbar type (horizontal or vertical), two integer arguments setting the minimum and maximum range values, and a Boolean flag directing the scrollbar to be redrawn (if `TRUE`).

```
SetScrollbarRange( hwnd, SB_VERT,  
                  nRangeMin, nRangeMax, FALSE );  
SetScrollbarPos( hwnd, SB_VERT, nScrollbarPos, TRUE );
```

After setting or resetting the scrollbar range, the scrollbar's thumbpad position still must be set. Again, `SetScrollbarPos` is called with the application window's handle and a constant identifying the scrollbar type—a horizontal or a vertical scrollbar—followed by the new position and a flag directing the scrollbar to be redrawn, or if `FALSE`, to be left as is.

The actual scrollbar range adjustment is handled in response to the `WM_SIZE` message (discussed a bit later in the chapter, in the “Windows Sizing and Resizing” section). The scrollbar thumbpad position is updated regularly in response to `WM_VSCROLL` or `WM_HSCROLL` messages.

For an MFC-based application, the process is essentially the same, except that the `SetScrollbarRange` and `SetScrollbarPos` functions are `CWnd` class methods and do not require the window handle. On the other hand, both the `SetScrollbarRange` and `SetScrollbarPos` functions are internal when using the `CScrollView` class, so the `SetScrollbarSizes` function is used to set the scrollbar ranges as well as the scrollbar page and line step sizes.

Scrollbar Message Handling

Because the `SB_XXXXXX` messages posted by scrollbar events are secondary messages, a standard Windows application begins by looking for either a `WM_VSCROLL`

or `WM_HSCROLL` message, indicating that the mouse event occurred in the vertical or horizontal scrollbar, respectively. The secondary event message is found in the `wParam` value.

For an MFC-based application, the equivalent is to use the ClassWizard to create two methods: `OnHScroll` and `OnVScroll` in the `CPainText2View` class. Here, however, instead of `wParam` and `lParam` values, three arguments are supplied. Two of these arguments are derived from the standard `wParam` and `lParam` arguments, identifying the type of scroll message and the thumbpad position on the scrollbar. The third argument is a pointer to the `CScrollbar` class instance originating the message.

Because the horizontal and vertical scrollbar responses are quite similar (in the *PainText* demo), the `WM_VSCROLL` message handling is used here to illustrate both cases, beginning as:

```
case WM_VSCROLL:
    switch( LOWORD( wParam ) )
    {
```

Under earlier Windows versions, the `wParam` argument was commonly accessed as `switch(wParam)` without requiring the `LOWORD` macro. In Windows 98, however, the `wParam` argument has changed from a 16-bit to a 32-bit argument. Despite this change, the secondary message value accompanying a `WM_COMMAND`, `WM_VSCROLL`, or `WM_HSCROLL` message (among others) is still a 16-bit value but is now passed as the low word in `wParam`. In many cases, because the high-word value will be `NULL` or zero, `switch/case` handling will function without including the `LOWORD` reference. But you can't depend on this. Ergo, the `LOWORD` macro should always be used to explicitly extract the 16-bit secondary message from the 32-bit argument.

In an MFC version, if you were handling the scrollbar events directly, they would already be identified in the `nSBCode` parameter and handled as:

```
switch( nSBCode )
{
```

NOTE

Remember, the *PainText2* application uses the `CScrollView` class and, therefore, in this demo, these functions are handled internally and do not appear in the source code.

Because the order in which the scrollbar events are handled is not important, the first two handled here will be the endpad messages: SB_LINEUP and SB_LINEDOWN.

```
case SB_LINEUP:
    cyStep = -1;
    break;

case SB_LINEDOWN:
    cyStep = 1;
    break;
```

Both vertical and horizontal scrollbar cases provide essentially the same response, using a set step value for vertical movement of one line up or down; for the horizontal scrollbar, the equivalent would be a character movement left or right. And, yes, for the present, the graphics text is being treated very much like a row/column text display.

Alternatively, the scrollbar positions themselves could be incremented or decremented at this point. But, for the moment, it's simplest to set a variable at this point and then later adjust the display and scrollbar positions appropriately.

The next two events are the SB_PAGEUP and SB_PAGEDOWN messages:

```
case SB_PAGEUP:
    cyStep = min( -1, -cyWin / cyChr );
    break;

case SB_PAGEDOWN:
    cyStep = max( 1, cyWin / cyChr );
    break;
```

In these two instances, the movement range is calculated from the size of the client window (cyWin) and the vertical line spacing (cyChr), with a simple range check to return a minimum line adjustment of 1 (or -1).

The line and page scroll messages are relatively simple, but the SB_THUMBTRACK message requires a different provision. For this event, instead of an incremental adjustment, the scroll step size is the differential between the stored scrollbar position (cyPos) and the new position reported in the low word of the lParam argument.

```
case SB_THUMBPOSITION:
    cyStep = LOWORD( lParam ) - cyPos;
    break;
```

Remember, each scrollbar has already been assigned a range for full-scale movement. Windows, in the low word of `lParam`, is simply reporting the relative position on the assigned scale. At this point, the application is calculating the differential, so presently another calculation can be made to move the thumbpad to the same position that was reported. Slightly inefficient, isn't it?

Fortunately, execution efficiency isn't important in this particular series of routines. What is important—a smooth, seamless response to dragging the thumbpad—is being accomplished efficiently with a minimum of source code. As always, it's a choice of trade-offs.

If you've realized that no provisions have been made to track the thumbpad—the `SB_THUMBPOSITION` event repositions the thumbpad after the mouse button is released and not while it's being dragged—this is a good time to explain that these two forms are generally treated as alternatives, not complements.

Continuously tracking the thumbpad position has one major flaw: It tends to be sluggish, particularly if responding to a change in position requires much calculation or screen activity. Therefore, many applications prefer to use the `SB_THUMBPOSITION` message and ignore the `SB_THUMBTRACK` messages. However, if you wish, the following fragment can be implemented to provide thumbpad tracking:

```
case SB_THUMBTRACK:
    cyStep = LOWORD( lParam ) - cyPos;
    break;
```

For a comparison, try holding the button down on the scrollbar track, generating a series of `SB_PAGEDOWN` or `SB_PAGEUP` messages. Then execute a similar scroll using the `SB_THUMBTRACK` response. Of course, the vertical length of the display is also a factor, but overall, the differences are distinct.

Last, a default case is provided purely as a precaution to reset the step value:

```
default:
    cyStep = 0;
    break;
```

Before you decide that the default case is redundant, consider for a moment the results if, for example, a `SB_PAGEDOWN` event is followed by a series of unrecognized `SB_THUMBTRACK` messages, without resetting `cyStep`.

Cautionary tales aside, the rest of the story is found after the `switch/case` statement finishes.

```
if( cyStep = max( -cyPos, min( cyStep, cyMax - cyPos ) ) )
{
    cyPos += cyStep;
}
```

The first provision is simply a range check, after which the scrollbar position variable is incremented according to the `cyStep` value (which may be a negative or positive integer).

Having reset the `cyPos` variable, the next requirements are to adjust the window position to match and then update the position of the scrollbar's thumbpad.

```
ScrollWindow( hwnd, 0, -cyChr * cyStep, NULL, NULL );
SetScrollPos( hwnd, SB_VERT, cyPos, TRUE );
UpdateWindow( hwnd );
}
```

And, last, the `UpdateWindow` function is called to ensure that the client window is repainted. This results in a `WM_PAINT` message being posted and requires its own response.

In responding to the `ScrollWindow` function, Windows has set the invalidate rectangle coordinates to cover the area revealed by the scroll operation. Therefore, the subsequent `WM_PAINT` operation is required to paint only a portion of the screen, which is faster than repainting the entire client window.

Window Sizing and Resizing

While the *PainText* demo is deliberately designed to create a display larger than the actual client window, thus necessitating the use of scrollbars, further provisions are also required to respond to changes in the size of the client window.

Any time the client window changes size—vertically or horizontally, larger or smaller—a `WM_SIZE` message is posted to the application. Also, when the application was first created, a `WM_SIZE` message preceded the initial `WM_PAINT` message.

In the *WinHello* demo, the `WM_SIZE` message is left for default handling by Windows instead of being handled by the application itself. The *PainText* application, however, is intended to be a bit more sophisticated in its response, and for this purpose, several operations are necessary.

Size Message Handling

In response to a `WM_SIZE` message, the first requirement is to retrieve the new `cyWin` and `cxWin` values that accompany the `WM_SIZE` message as the high-word and low-word values in the `lParam` argument, thus:

```
case WM_SIZE:
    cyWin = HIWORD( lParam );
    cxWin = LOWORD( lParam );
```

In the MFC version, the `OnSize` function, which is the response to a `WM_SIZE` message, is called with three parameters: the `nType` argument, which reports the type of sizing operation but which will be ignored here, and the `cx` and `cy` arguments, which report the new client window size.

```
void CPainText_View::OnSize( UINT nType,
                             int cx, int cy)
{
    CView::OnSize(nType, cx, cy);
    // TODO: Add your message handler code here
    if( m_cxChr > 0 && m_cyChr > 0 )
    {
        // don't do anything unless have text sizes
        m_cyWin = cy;
        m_cxWin = cx;
```

NOTE

Again, in the *PainText2* demo, the `OnSize` method is internal to the `CScrollView` class and no direct handling is required.

Before responding to the size-change message, the first step is to determine if you have vertical and horizontal character sizes. Using the MFC classes, the sequence of size messages is not quite as clean as it is in a conventional application. A size event will be reported before the system text metrics have been retrieved—depending, of course, on which version is used to test the text metrics. However, to prevent a runtime error caused by a divide-by-zero operation, a simple test is provided.

Also, in the MFC version, the size information is stored in member variables for the `CPainText_View` class rather than in global variables. The effect is the same, but the mechanism is slightly different.

Scrollbar Adjustments

Once you know the new window size, set the vertical and horizontal scrollbars to match. Begin by calculating a new value for `cyMax` and take into account the number of lines that can be displayed in the resized window.

```
cyMax = max( 0, NUM_LINES + 2 - cyWin / cyChr );  
cyPos = min( cyPos, cyMax );
```

In like fashion, the scrollbar thumbpad position is also recalculated before next resetting the scrollbar range and thumbpad position.

```
SetScrollRange( hwnd, SB_VERT, 0, cyMax, FALSE);  
SetScrollPos(  hwnd, SB_VERT,  cyPos, TRUE );
```

The MFC version follows essentially the same pattern.

And last, a similar treatment is accorded the horizontal scrollbar. In theory, and according to various documentation, the scrollbar ranges should only require adjusting when the application window is created or resized. Experience, however, has suggested that better results are achieved if the scrollbar range is reset immediately before adjusting the thumbpad position.

In later examples in this book, the text metrics operations demonstrated by the *PainText* program will be used in a variety of other applications, as will the scrollbar-handling provisions and the text-handling and positioning routines.

Simplified Operations with MFC

The *PainText2* demo shows a much simplified method of presenting information in a scrollable format by using the `CScrollView` class as the basis for the display window. In the *PainText2* demo, instead of the application writing only the information that will fit in the display window, a virtual space is supplied where the entire document can be written. Then, after the document is written, the `CScrollView` supplies all of the scroll operations.

This does not mean that a monstrous copy of the entire document is kept in memory as an image. As I'm writing this chapter, more than 40 pages of text are actively available, but only approximately a half-page is visible in the display window. In other cases, I may be working on a document containing several hundred pages, again with only a half-page or so visible.

And, remember, because that half-page—at my display resolution—requires roughly 2MB of memory to display, the entire document could demand 80MB to 90MB of memory (as an image).

Instead, the `CScrollView` class simply controls which part of the drawing operation is actually sent to the display context for viewing, but it allows the drawing operation to act as if it were writing everything. Furthermore, the `CScrollView` class manages the scrollbars for the view window, intercepts and interprets the scrollbar messages, and invisibly handles most of the operations demonstrated in the *PainText* demo.

Setup for CScrollView

Using the `CScrollView` class does, however, have its own requirements. First, because the information used for the display is artificially structured, you need to retrieve some information about the text metrics (the font size characteristics). For the *PainText2* demo, this retrieval was shown earlier in the `OnShowWindow` method (see “Text Sizing”).

Once you have retrieved the font information, you use the `OnUpdate` method to provide four important pieces of information to the `CScrollView` class.

The first and obligatory piece of information is the mapping mode to be used in the `CScrollView` window. This argument may be any of the Windows mapping modes except `MM_ISOTROPIC` or `MM_ANISOTROPIC`. To use an unconstrained mapping mode, you should call `SetScaleToFitSize` instead of `SetScrollSizes`.

The second and the one really essential piece of data is the x/y size of the virtual window where you want to write your information. This pair of size coordinates does not restrict what you write (or draw), but it does place limits on how you can scroll across the virtual window. If the specifications are too small, you will simply not be able to view all of the contents. If the specifications are too large, the view will be able to scroll beyond the displayed data.

The third and fourth pieces of data are optional arguments and set the x/y scroll sizes for page and line steps, respectively. If not supplied, they will default to predefined step sizes.

The following code fragment, from the *PainText2* demo, shows one implementation.

```

void CPainText2View::OnUpdate( CView* pSender, LPARAM lHint,
                              COBJECT* pHint)
{
    if( m_cxChr > 0 && m_cyChr > 0 )
        SetScrollSizes( MM_TEXT,
                        CSize( m_cxChr*100, m_cyChr*100 ),
                        CSize( m_cxChr*10, m_cyChr*10 ),
                        CSize( m_cxChr, m_cyChr ) );
    else
        SetScrollSizes( MM_TEXT, CSize( 1300, 2000 ),
                        CSize( 130, 200 ), CSize( 13, 20 ) );
}

```

Here, two different provisions are offered. The first depends on font information to calculate the `CScrollView` window size and scroll step sizes; the second uses predefined sizes if the font information is not available.

In other circumstances—for example, when you wish to show a document—you may be less concerned with font information and more interested in document-size information. For this purpose, when implementing your custom document class, you would include a function to return a `CSize` response giving the size of the document, such as `GetDocSize()`.

The OnDraw Method

Having set up the `CScrollView` class instance, the remaining task is painting the actual window display. In the *PainText2* demo, this is similar to the original *PainText* demo, but not identical. The *PainText2* version uses the `OnDraw` method, as shown here:

```

void CPainText2View::OnDraw(CDC* pDC)
{
    CPainText2Doc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    CString    csText;
    int        i, x, y;
    TEXTMETRIC tm;

    pDC->GetTextMetrics( &tm );
    m_cyChr = tm.tmHeight;
    m_cxChr = tm.tmAveCharWidth;
    pDC->SetTextAlign( TA_LEFT | TA_TOP );
    for( i=0; i<=NUM_LINES; i++ )

```



```
{
    x = 1 + m_cxChr * i;
    y = 1 + m_cyChr * i;
    csText.Format(
        "This is line %d being displayed at X:%d / Y:%d",
        i, x, y );
    pDC->TextOut( x, y, csText );
}
```

Notice that the `OnDraw` method in *PainText2*, unlike the `WM_PAINT` response in *PainText*, does not calculate a beginning or ending line number and that no scrollbar positions are used to calculate offsets. Instead, the entire display is being written just as if the window were actually large enough to display all of the information. In turn, the `CScrollView` window provides the view and offsets into the virtual space.

Differences in the Demos

To summarize, you should note the following differences between the *PainText* and *PainText2* (MFC) examples:

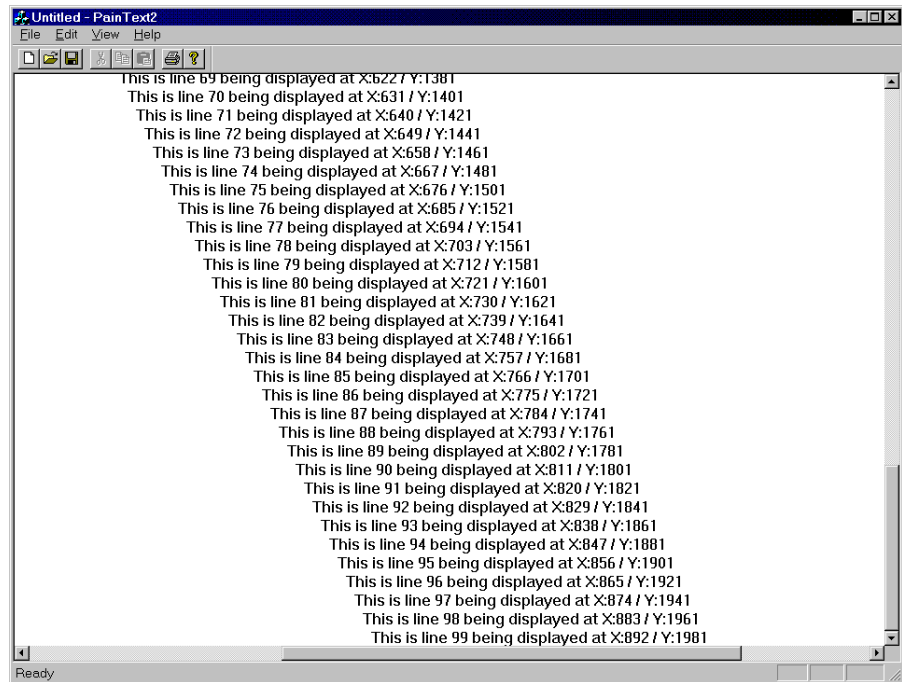
- In the `CScrollView`-derived window, no efforts are made to track the scrollbar positions or to offset the drawing information accordingly.
- No limits are placed on where and what information can be drawn.
- Window size information is required.

Also, if you compare Figure S1.4 with Figure S1.2, you should see the differences in the line offsets.

In Figure S1.2, the line offsets shown are relative to the window. In Figure S1.4, the line offsets are relative to the document (the offsets shown are relative to a much larger space than the space shown in the visible `CScrollView` window). By itself, this factor is a powerful argument for using the `CScrollView` class. By drawing in a virtual space of whatever size desired rather than trying to draw portions of an image to fit within a view window, the task of constructing a complex image or document is greatly simplified.

FIGURE S1.4:

Multiple text lines in
PainText2



This chapter described how to implement several provisions required by most Windows applications: writing a display larger than the application window, scrolling the display within the window, and responding to several event messages. We showed how these operations are done with conventional programming, in the *PainText* demo, and how they are simplified by using MFC, in the *PainText2* demo.

In the next chapter, we will address another requirement for Windows applications: responding to keyboard events.

S U P P L E M E N T

T W O

S2

Keyboards, Carets, and Characters

- Keyboard-event message components
- Virtual key codes and handling
- Keyboard message responses and handling
- Text caret (cursor) handling
- Event message generation

The standard 89-key and the enhanced 101–102-key keyboards, as well as the newer one-handed key encoders, are the input devices of choice for today’s computer user. And even though all types of applications—not just Windows 98 programs—are becoming increasingly mouse-interactive, until such time as either mechanical telepathy or direct neural interfaces become common and reliable, the keyboard will remain the primary system input device. Yes, I know that pen-based computers are out there. But can we really take them seriously? Not with my handwriting!

Because the keyboard is so important, knowing how Windows 98 handles keyboard events is critical. The different languages and character sets used around the world also require different programming considerations. This chapter will show you how to manage keyboard-events in Windows applications.

The Evolution of Keyboard Character Sets and International Language Support

Originally, PCs recognized a single character set with the character values 0x20 through 0x7E devoted to the English alphabet, the Roman number set, and assorted punctuation. The remaining 128 character values—0x80 through 0xFF—were devoted to the extended ASCII characters, providing a selection of Greek and mathematical symbols and a primitive series of box-drawing characters.

On the whole, this character set was strictly English-chauvinistic. For non-English writers, it provided an exercise in frustration, as they attempted to render their native languages via a restrictive display channel. And, while various approaches attempted to circumvent these limitations, these often innovative experiments have become, principally, a matter of historical interest only.

Today, since computers are international in both nature and recognition, a variety of keyboard drivers (with corresponding video and printer character sets) provide support for most, if not all, of the principal languages. In some cases, such as the Japanese Kanji alphabet, this support may involve special keyboard layouts. In other cases, such as the Thai alphabet, the familiar keyboard can be adapted (with only a change of key caps for Thai characters); only special support for the display format is required, along with an appropriate driver, of course.

Earlier versions of Windows provide international alphabets, but the 256 characters supported by an 8-bit character code, such as the familiar ANSI ASCII standard, are not adequate for true international support. For this reason, a new standard, called *Unicode*, has been defined. In Unicode, characters use a 16-bit char (or wide char) code to support a total of 65,536 characters—quite sufficient to encode all of the world’s contemporary alphabets.

Of course, this does not mean that every keyboard will be required to support the entire Unicode char set. Instead, keyboard drivers access subsets of the Unicode standard for specific languages. Win32 (Windows 98) applications have the options of supporting either Unicode or the conventional ASCII character set or providing mixed support for both.

Although you don’t need to worry about Unicode in most cases, sometimes it is important for international support, even on an introductory level. The *KeyCodes* and *Editor* demos, which are discussed in this chapter, demonstrate this support.

ANSI versus MBCS versus Unicode

The conventional ANSI character set consists of an 8-bit font (255 characters) and is limited to the Roman alphabet together with some European variations and an assortment of symbols. The shortcoming of the ANSI character set is that it limits displays to languages using the Roman alphabet—which also means that a large part of the world’s population is not able to view computer displays in their native language(s).

Although multibyte characters (MBCS) have been used for a number of international languages, the multibyte approach has never been standardized; it often requires specific firmware and software while still leaving the operating system itself limited to an English (or Roman-alphabet) display. One example of the MBCS approach is in the Japanese Kanji script.

In contrast, Unicode characters—a.k.a. “wide characters”—use 16-bit character descriptors (for 65,535 characters) and include character sets for every language used in the modern world, including Kanji, as well as technical symbols and special publishing characters.

Admittedly, wide characters do take more space in memory than multibyte characters but they are processed faster. Another disadvantage to multibyte characters is that only one locale can be implemented at a time when using multibyte encoding, but Unicode representation encompasses all (terrestrial) character sets.

Continued on next page

Both MFC and the runtime library support Unicode or MBCS. (The MFC framework is Unicode-enabled throughout, except for the database classes; ODBC is not presently Unicode-enabled.)

Both Unicode and MBCS are enabled by means of portable data types in MFC function parameter lists and return types. These types are conditionally defined in the appropriate ways, depending on whether your build defines the symbol `_UNICODE` or the symbol `_MBCS` (which means *DBCS* or double-byte character set). Different variants of the MFC libraries are automatically linked with your application, depending on which of these two symbols your build defines.

Class library code uses portable runtime functions and other means to ensure correct Unicode or MBCS behavior.

Most traditional C and C++ code makes assumptions about character and string manipulation that do not work well for international applications; however, the application must still handle certain kinds of internationalization tasks in the application code. To simplify internationalization, always:

- Use the same portable runtime functions that make MFC portable under either environment.
- Make literal strings and characters portable under either environment using the `_T` macro.
- Follow precautions for MBCS strings during parsing; these precautions are not required under Unicode.
- Mix both ANSI (8-bit) and Unicode (16-bit) strings in an application if needed, but do not mix them in the same string.
- Never hard-code strings in any application. Instead, strings should always be `STRINGTABLE` resources in the application's .RC file. This allows an application to be localized without changing the source code and recompiling.

You can find more extensive information on Unicode versus ANSI character sets in the online documentation included with your compiler.

Handling Keyboard Events

One of the principal strengths of the Windows environment is its international adaptability. During installation, or via the International dialog box in the Control

Panel, you can select a variety of keyboard drivers, along with fonts and .DLL libraries that support various international keyboard configurations and character sets.

If you are reading this in English, most likely the keyboard drivers available with your version of Windows 98 are principally those suited to the Germanic/Romance languages. If you are reading a translated version of this book, your version of Windows 98 probably supplies a different selection of supported keyboards and alphabets.

No matter which keyboard driver you're using, Windows takes control of your keyboard and actively traps all keyboard events, long before they could be received and interpreted by an application. Routines supplied by Windows decode these key events, storing the results as keyboard-event messages in the application's message queue; Windows decides which message queue the keyboard input is intended for.

Then, the target application needs to retrieve the keyboard-event messages that Windows has stored in its message queue. When control returns to the target application, the `GetMessage` function (refer to the `WinMain` routine discussed in Chapter 2) retrieves the keyboard-event message from the queue. After it is retrieved, the application treats the event message as characters received directly from the keyboard, in a fashion similar to how the event message is handled under DOS.

However, you need to be careful because *similar* does not mean *identical*. The principal difference between a DOS application receiving keyboard-event messages and a Windows application retrieving keyboard-event messages from the application's message queue is primarily the amount of information that is reported.

Early computers, with their slow CPUs and limited memory, couldn't handle complex information from the keyboard without sacrificing what little speed of execution they were capable of. (Some very early machines did not even recognize, except via special provisions, a difference between uppercase and lowercase characters and handled only 6-bit char codes.) For these systems, the wealth of information available through a modern keyboard would be overwhelming.

Under DOS, the keyboard-event message is essentially the scan and char codes returned by the keyboard buffer. When the user presses the *a* key, a DOS keyboard buffer is fed only the single character *a* returned as the char code. The scan code, which identifies the physical key and some aspects of the Shift+Ctrl+Alt

flag states, has already been discarded as unnecessary. (For details, see DOS interrupts 16h and 21h.)

But Windows 98 and, to a lesser degree, earlier versions of Windows are designed for operation on less limited and less time-constrained CPUs. Under Windows, the same physical event of pressing the *a* key begins by reporting three separate event messages:

- A `WM_KEYDOWN` event reporting that a key is being pressed
- A `WM_CHAR` event reporting the character code generated (in the `wParam` argument)
- A `WM_KEYUP` event reporting the subsequent key release (unless, of course, the key is held down for auto-repeat)

Each of these three messages is accompanied by `wParam` and `lParam` arguments. The `WM_CHAR` event comprises six separate information fields. Ergo, a total of eight data elements are reported for each message received, resulting in a grand total of 24 pieces of data generated by one keystroke!

Fortunately, applications are not required to use all of the information or even to recognize its existence, so you may feel free to wipe the worried sweat away from your brow and cease considering the keyboard as a potential instrument for hari kari. All of the information is there when needed, but it is not obligatory.

Types of Keyboard-Event Messages

Each keyboard event begins by generating a window message indicating that the event is one of two things: an *application message* or a *system keystroke message*. Application messages are `WM_KEYDOWN`, `WM_KEYUP`, `WM_CHAR`, and `WM_DEADCHAR`. System keystroke messages are `WM_SYSKEYDOWN`, `WM_SYSKEYUP`, `WM_SYSCHAR`, or `WM_SYSDEADCHAR`.

System keystroke messages, identified as `WM_SYSxxxxx`, are generally events that are more important to Windows than to the application. Normally, they are simply passed by default to `DefWindowProc` for handling. However, the *KeyCodes* demo described in this chapter traps and displays almost all keyboard-event messages, both system and application, together with all of the information accompanying each.

TIP

Because the system interprets the event message as requiring special actions, Windows, in some cases, takes control after the first part of the key event is processed. Thus, a few keyboard events are only partially trapped by this program. The reasons for such event instances should be readily apparent to any programmer even partially familiar with the Windows 98 interface. But, if you are not familiar with the interface, the discontinuity occurs simply because this program loses the input—and, therefore, the subsequent key events—whenever a hotkey combination redirects operations to another application or utility.

System-key event messages include Alt-key combinations, such as the Alt+Tab or Alt+Esc combinations, which are used to switch the active window or the system menu accelerators. (Alt-key combinations are discussed in detail in Supplement 9.) In some cases, these event messages may be used directly by applications, but they are normally left for Windows to recognize and handle.

In like fashion, the application ignores both the application and system `xxxDEADCHAR` messages. In general, these are used by non-US keyboards that include special keys for adding accents or diacritics to other letters. They can also provide other special functions or, in Unicode applications, produce other special selections. Because these messages do not, of themselves, generate characters, they are called “dead” and, for most applications, they can be safely ignored.

Similarly, the `WM_KEYUP`, `WM_SYSKEYUP`, `WM_KEYDOWN`, and `WM_SYSKEYDOWN` messages can generally be ignored by the application. If circumstances demand, however, the information held in these messages is available.

NOTE

Sometimes, such as when the Shift key is used to allow multiple selections, it's more important to know when a key has been released than which key was pressed. Or, under other circumstances such as a game, you might be interested in reaction times and want to know how long a key was held down by checking both the key down and key up events and the time stamps for each. If you have special requirements, this information is available even if it isn't normally used by most applications.

After all of these exclusions, the `WM_CHAR` message remains. This event provides the single keyboard message that applications normally process because it contains the actual character code identifying the key event.

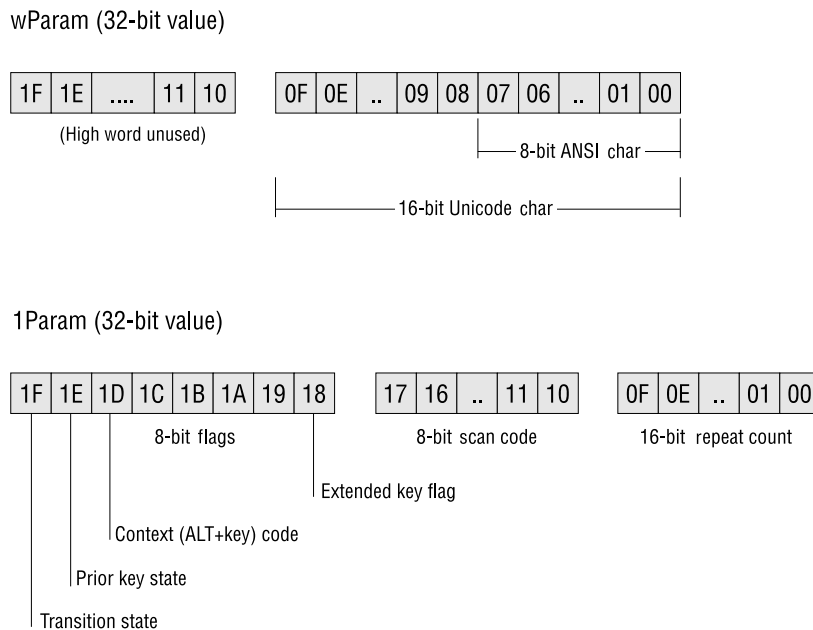
Elements of Keyboard Messages

In DOS, in addition to the character and scan codes, keyboard shift-state information is also available, and this information can be retrieved separately from the keyboard event itself. The DOS keyboard shift-state information contains the current status of the right and left Shift keys, the right and left (assuming an enhanced keyboard) Ctrl and Alt keys, the CapsLock and NumLock keys, and the Scroll Lock and Insert keys (which are usually ignored). Under DOS, however, key-press and key-release events are not normally reported as separate events.

In Windows, the `wParam` and `lParam` arguments accompanying each keyboard-event message carry the event character code, the scan code, all of the shift-state information just mentioned and, in addition, an 8-bit key repeat count. Figure S2.1 illustrates the components of the `wParam` and `lParam` arguments for `WM_CHAR`.

FIGURE S2.1:

`WM_CHAR`'s `wParam` and `lParam` arguments



The wParam Argument

In previous versions of Windows, the character code was found in the low byte of the `wParam` argument.

Under Windows 98, the `wParam` argument has grown to 32 bits, and the character code can be either the low byte of the low word (for 8-bit ANSI ASCII values) or, for 16-bit Unicode char values, the low word itself. Which type of character code is expected is determined by compiler switch settings or by direct references to Unicode/ANSI API functions.

The high-word value in the `wParam` argument currently remains unused.

The lParam Argument

The `lParam` argument contains a variety of information, as explained in the following sections.

Repeat Count This 16-bit value reports the number of keystrokes represented by the event message and, normally, has a value of 1. However, if a key is held down and the application cannot process the keyboard messages quickly enough (for any reason), Windows will combine multiple `WM_KEYDOWN`, `WM_SYSKEYDOWN`, `WM_CHAR`, and `WM_SYSCHAR` messages into a single message, incrementing the repeat count appropriately. For `WM_KEYUP` or `WM_SYSKEYUP` messages the repeat count is always 1, of course.

One reason that the repeat count value may be higher than 1 is as a response to a *typematic* overrun. For example, you may observe typematic overruns while executing the *KeyCodes* demo, even on fast machines, simply because of the time required to write a full-line screen message for each key event. You might also generate typematic overruns by selecting a high repeat rate from the Control Panel. In neither case, however, should you take this as an indication that typematic overruns are likely in conventional applications.

In addition, the Page Up, Page Down, and up and down arrow keys are sometimes sources for typematic overruns, simply because the time required to respond to one instruction may be long enough for additional keystrokes to accumulate and, as a result, often cause applications to overscroll in response. For this reason, many applications prefer to ignore the repeat count value, particularly for specific keys. Experiment to determine whether individual applications should use or ignore the repeat count.

Scan Code The keyboard scan code is a value generated by the keyboard firmware to identify the actual physical key pressed or released. While the same char code may be generated by different keys—for example, the numeral 1 character can be returned either from the number key at the top of the main keyboard or from the key pad—the scan code is specific to the physical key.

Likewise, right and left Alt and Ctrl keys return separate scan codes, even though neither generates a char code. However, some physical keys may return different scan codes according to the Alt, Ctrl, or Shift states.

Extended Key Flag This flag is set to 1 if the present keystroke was generated by one of the keys specific to the enhanced keyboard. These keys include the nonkeypad cursor and page keys, as well as the Insert and Delete keys, the keypad slash key (/), the keypad Enter key, and the NumLock key.

Context Code Flag This flag is set to 1 if the Alt key is down during the present keystroke or if the current message is WM_SYSKEYUP or WM_SYSKEYDOWN. The context code flag is cleared (set to 0) for all WM_KEYUP and WM_KEYDOWN messages with two exceptions:

- Some non-English keyboards use combinations of the Shift, Ctrl, and Alt keys together with conventional keys to generate special characters. These may have the context code flag set but will not be reported as system keystrokes.
- If the active window is an icon, it does not receive the input focus, and, therefore, all keystrokes will generate WM_SYSKEYUP and WM_SYSKEYDOWN messages to prevent the active window (as an icon) from attempting to process these events. In these cases, the context code flag is set only if the Alt key is down.

Prior Key State Flag For WM_CHAR, WM_CHARDOWN, WM_SYSCHAR, and WM_SYSCHARDOWN messages, the prior key state flag is set to 1 if the same key was previously down, or it is cleared (set to 0) if the same key was previously up.

For WM_KEYUP and WM_SYSKEYUP messages, the prior key state flag is always set. Obviously, the key must have been down before it could be released.

Transition State Flag This flag provides redundant information. It is cleared (0) if a key is being pressed, as in a WM_KEYDOWN or WM_SYSKEYDOWN message. If the key is being released, as in a WM_KEYUP or WM_SYSKEYUP message, the flag is set to 1. For the WM_CHAR and WM_SYSCHAR messages, the transition flag is cleared (but it's also irrelevant).

The KeyCodes Demo: Deciphering Keyboard-Event Messages



The *KeyCodes* and *KeyCodes2* demos provide a window in which to examine keyboard-event messages in a degree and detail that are not available through conventional handling. Figure S2.2 illustrates a series of key-event messages captured by the *KeyCodes2* demo.

FIGURE S2.2:

Keyboard-event messages deciphered through the *KeyCodes2* program

Message	Code	Key	Char	Cnt	Scan	Ext	ALT	Prv	Trs
WM_KEYUP	.20h			1	39h		*	*	*
WM_KEYDOWN	.54h			1	14h				
WM_CHAR			[t]	1	14h				
WM_KEYUP	.54h			1	14h		*	*	*
WM_KEYDOWN	.48h			1	23h				
WM_CHAR			[h]	1	23h				
WM_KEYDOWN	.45h			1	12h				
WM_CHAR			[e]	1	12h				
WM_KEYUP	.48h			1	23h		*	*	*
WM_KEYUP	.45h			1	12h		*	*	*
WM_KEYDOWN	.20h			1	39h				
WM_CHAR			[]	1	39h				
WM_KEYUP	.20h			1	39h		*	*	*
WM_KEYDOWN	.41h			1	1Eh				
WM_CHAR			[a]	1	1Eh				

The *KeyCodes* demo uses eight case statements to respond to the keyboard WM_XXXX messages. The ShowKey subprocedure provides the mechanism to decipher and expand the keyboard-event messages. Let's look at a few provisions in this example that deserve some explanation.

NOTE

The *KeyCodes* and *KeyCodes2* demos are included on the CD that accompanies this book, in the Supplement 2 folder.

Retrieving Text Metric Information

The first provision of interest is in the `WndProc` procedure. In response to the `WM_CREATE` message, the following code fragment is used to retrieve text metric information for the system fixed (monospace) font:

```
case WM_CREATE:
    hdc = GetDC( hwnd );
    SelectObject( hdc, GetStockObject( SYSTEM_FIXED_FONT ) );
    GetTextMetrics( hdc, &tm );
    cxChr = tm.tmAveCharWidth;
    cyChr = tm.tmHeight + tm.tmExternalLeading;
    ReleaseDC( hwnd, hdc );
    rect.top = 2 * cyChr;
```

The `tm` (text metric) data structure is used to retrieve information about the system fixed font. The character size (in pixels) is then used to set the `cxChr` and `cyChr` variables, which will be used to control the format and spacing of the actual display.

NOTE

`SYSTEM_FIXED_FONT` is only one of the many fonts available under Windows 98 (fonts will be discussed in Supplement 14). I choose it for the demo because it provides a simple, monospace font that is available on all (English/US) systems (and probably on most others as well).

The second, parallel provision is found in the response to the `WM_PAINT` message and in the `ShowKeys` subprocedure. Before text is written, the same `SelectObject` and `GetStockObject` functions are called again. But this time, instead of being called in an information context, they are called as the display font.

```
InvalidRect( hwnd, NULL, TRUE );
hdc = BeginPaint( hwnd, &ps );
SelectObject( hdc, GetStockObject( SYSTEM_FIXED_FONT ) );
...
EndPaint( hwnd, &ps );
```

TIP

Remember, if you arrange spacing to fit a specific font, you also need to be sure that the same font is used for the actual display (unless you aren't particularly concerned about the results).

Responding to Keyboard Messages

Another part of the *Keycodes* demo that deserves some explanation is the `switch(msg)`... tree, which provides eight case responses for the eight keyboard-event messages tracked:

```
switch( msg )
{
    ...
    case WM_KEYDOWN:
        ShowKey( hwnd, 0, FALSE, "WM_KEYDOWN",      wParam, lParam );
        break;

    case WM_KEYUP:
        ShowKey( hwnd, 0, TRUE,  "WM_KEYUP",        wParam, lParam );
        break;

    case WM_CHAR:
        ShowKey( hwnd, 1, FALSE, "WM_CHAR",          wParam, lParam );
        break;

    case WM_DEADCHAR:
        ShowKey( hwnd, 1, FALSE, "WM_DEADCHAR",       wParam, lParam );
        break;

    case WM_SYSKEYDOWN:
        ShowKey( hwnd, 0, FALSE, "WM_SYSKEYDOWN",     wParam, lParam );
        break;

    case WM_SYSKEYUP:
        ShowKey( hwnd, 0, TRUE,  "WM_SYSKEYUP",        wParam, lParam );
        break;

    case WM_SYSCHAR:
        ShowKey( hwnd, 1, FALSE, "WM_SYSCHAR",         wParam, lParam );
        break;

    case WM_SYSDEADCHAR:
        ShowKey( hwnd, 1, FALSE, "WM_SYSDEADCHAR",    wParam, lParam );
        break;
    ...
}
```

In each case, the response is to call the `ShowKey` subprocedure with the appropriate data for the display, as shown in Figure S2.2.

NOTE

The demo provides a degree of grouping by including an extra half-line space after each `WM_KEYUP` or `WM_SYSKEYUP` event. However, as you may observe, the `_KEYDOWN`, `_CHAR`, and `_KEYUP` messages for a specific key do not always appear in strict sequential order.

Formatting the Text

In the demo, the remaining provisions for string handling and formatting the text display are fairly straightforward C code, with the possible exception of the `TextOut` function call. This function appears in text as:

```
TextOut( hdc, cxChr, cyWin - step, szBuff,  
        wsprintf( szBuff, szFormat[iType],  
                  (LPSTR) szMsg, ...
```

In the `ShowKey` subprocedure, the `wsprintf` function is called with an explicit typecasting instruction (`LPSTR`) preceding the `szMsg` reference. `LPSTR` is defined as a far pointer to a string and is necessary because, in this instance, `szMsg` has already been passed as a local pointer reference from the `WndProc` procedure. For clarity—and to prevent a compiler warning message—the explicit redefinition is made from a local to a far pointer.

The remaining parameters, such as the two string references `szBuff` and `szFormat[]`, are already passed to `wsprintf` as the expected far pointer references and do not require explicit typecasting. However, when in doubt, including explicit typecasting does no harm, but its omission might.

Ignoring Keyboard Messages

The *KeyCodes* demo demonstrates intercepting and reporting keyboard-event messages, but it also shows the potential hazards in attempting to process every message received. As mentioned earlier, if an application (or system) is not fast enough to process every keyboard message generated, Windows will combine duplicate messages, incrementing the repeat count appropriately. This approach may seem useful, but keep in mind that processing all—or even

most—keyboard messages is neither necessary nor desirable. The reasons for this are threefold:

- The `WM_SYSxxxx` messages are intended for Windows, not the application, to handle. Except under special circumstances, they can safely be left for processing via the `DefMessageProc` call.
- The `WM_KEYDOWN` and `WM_KEYUP` messages are essentially duplicates, and again, except under special circumstances, either or both can be ignored entirely. If, however, they are actually needed, most applications will confine themselves to responding to `WM_KEYDOWN` events while ignoring `WM_KEYUP` events.
- Even when `WM_KEYDOWN` messages are recognized, they are generally confined to cursor and special key events, not to retrieving conventional character key events. For this latter purpose, only the `WM_CHAR` message should be expected.

In the *KeyCodes* demo, all of the keyboard-event messages are given equal weight—a treatment that most applications will not indulge in. In the *Editor* demo discussed a bit later in this chapter, for example, only the `WM_KEYDOWN` and `WM_CHAR` messages are trapped, leaving the remaining six keyboard-event messages for default handling.

TIP

Windows applications should not depend on special key combinations, which may not be supported by many keyboard drivers and/or physical keyboards (especially non-English versions). Granted, this prohibition does eliminate a number of favorite “tricks,” but there are alternatives to using special key combinations. Also, your application will benefit by not relying on deciphering complex key combinations, because the time required for processing keyboard messages can be greatly reduced.

Translating Character-Event Messages

Applications dealing with text input depend on `WM_CHAR` keyboard-event messages to provide character input (as shown in the *Editor* demo). However, the `WM_CHAR` messages may or may not correspond to the familiar ASCII codes. So, how do you ensure that your application receives the appropriate char code when a key is pressed?

Actually, in this respect, there is very little problem. To match the keyboard drivers generating these messages, Windows 98 also provides translation services that have been incorporated in all of the programming examples. As you may recall, the message loop in the `WinMain` procedure appears like this:

```
while( GetMessage( &msg, NULL, 0, 0 ) )
{
    TranslateMessage( &msg );
    DispatchMessage( &msg );
}
```

The `TranslateMessage` function provides the mechanism to convert keystroke-event messages into character messages. These messages are recognized and used by the application because the keyboard, per se, generates only keystroke information. Before this new keystroke information can be used, the keyboard driver must translate it into `WM_XXXKEYDOWN` and `WM_XXXKEYUP` messages.

This translation process derives the `WM_CHAR` and `WM_DEADCHAR` messages from `WM_KEYDOWN` events, and the `WM_SYSCHAR` and `WM_SYSDEADCHAR` messages from `WM_SYSKEYDOWN` events. The translation service also processes shift-status flags to generate uppercase and lowercase characters.

The important point to remember is that the `WM_CHAR` message, via the `TranslateMessage` function, provides character information. The `VK_XXXXX` message parameters provide all function, cursor, and special key data. (`VK_XXXXX` message parameters will be covered presently in the “Virtual Key Codes” section of this chapter.) Both of these methods are demonstrated by the *Editor* demo on the CD accompanying this book.

While the `TranslateMessage` function is an inherent part of all Windows application programs, this function is hidden in MFC-based applications, where you use another provision to examine keyboard events.

Using MFC to Handle Keyboard Messages

If you are using MFC and creating an application using the AppWizard, the conventional message loop is effectively hidden and unavailable. This does not mean, however, that you cannot trap all these message events in the fashion shown in the *KeyCodes* demo; it just means that a slightly different approach is required for interception.

The *Keycodes* application shows how to trap the keyboard events using the `PreTranslateMessage` member function in the `CKeycodesView` class. The keyboard-event message is received as an `MSG` structure, which is defined as:

```
typedef struct tagMSG
{
    HWND    hwnd;
    UINT    message;
    WPARAM  wParam;
    LPARAM  lParam;
    DWORD   time;
    POINT   pt;
} MSG;
```

The `MSG` fields are defined as follows:

hwnd Identifies the handle of the window whose window procedure receives the message.

message Provides the message number or identifier.

wParam Provides additional information about the message. The exact meaning depends on the value of the `message` member.

lParam Provides additional information about the message. The exact meaning depends on the value of the `message` member.

time Provides the time when the message was posted.

pt Provides the cursor position, in absolute screen coordinates, at the time the message was posted.

The original *KeyCodes* demo program is only concerned with the `message`, `wParam`, and `lParam` fields in the `MSG` structure and ignores the `hwnd`, `time`, and `pt` fields. The `pt` field would be useful if you were intercepting mouse messages, but it really isn't relevant in this example, which demonstrates handling events generated from the keyboard.

In the *Keycodes2* demo's `PreTranslateMessage` handler, the `pMsg` structure is handled in the same fashion as the message events are handled in the original *KeyCodes* program (the non-MFC version). In this case, the exception is that the `pMsg` structure is provided in place of separate parameters and, therefore, must

be decoded before passing the arguments needed to the `ShowKey` function for display.

```

BOOL CKeycodesView::PreTranslateMessage(MSG* pMsg)
{
    switch( pMsg->message )
    {
        case WM_KEYDOWN:
            ShowKey( 0, 0, "WM_KEYDOWN",      pMsg->wParam, pMsg->lParam );
            break;

        case WM_KEYUP:
            ShowKey( 0, 1, "WM_KEYUP",        pMsg->wParam, pMsg->lParam );
            break;

        case WM_CHAR:
            ShowKey( 1, 0, "WM_CHAR",         pMsg->wParam, pMsg->lParam );
            break;

        case WM_DEADCHAR:
            ShowKey( 1, 0, "WM_DEADCHAR",     pMsg->wParam, pMsg->lParam );
            break;

        case WM_SYSKEYDOWN:
            ShowKey( 0, 0, "WM_SYSKEYDOWN",   pMsg->wParam, pMsg->lParam );
            break;

        case WM_SYSKEYUP:
            ShowKey( 0, 1, "WM_SYSKEYUP",     pMsg->wParam, pMsg->lParam );
            break;

        case WM_SYSCHAR:
            ShowKey( 1, 0, "WM_SYSCHAR",      pMsg->wParam, pMsg->lParam );
            break;

        case WM_SYSDEADCHAR:
            ShowKey( 1, 0, "WM_SYSDEADCHAR",  pMsg->wParam, pMsg->lParam );
            break;
    }
    return CView::PreTranslateMessage(pMsg);
}

```

Notice also that all of the events are returned for default handling by the parent class's `PreTranslateMessage` function. If we had not defined a custom `PreTranslateMessage` handler here, this would have been the default handling provided by MFC.

NOTE

Chances are that you will not need to provide this type of handling for any of your own applications. For most key events, MFC provides alternatives that do not need this level of interrogation. Nevertheless, you are still able to intercept keyboard (and other event) messages at the lowest possible level.

Handling Virtual Keys

Under DOS, many applications, particularly TSRs, spend a portion of their time filtering keyboard char codes while waiting for a specific keyboard event to trigger some action. This can be as complex as looking for a Ctrl+Alt+K combination or as common as waiting for an arrow or page key. Hotkey assignments under Windows 98 are not handled in the same fashion as DOS TSRs, nor should they be. In some ways, Windows can be thought of as a giant TSR parceling out keyboard-event information to applications requesting that information.

NOTE

DOS-based TSRs commonly install vectors to intercept all keyboard inputs. After checking each keystroke, the TSR returns the data to the original location so that it can be sampled by the next TSR and, finally, sent to the principal application.

Virtual-key handling is one area where applications moving from DOS to Windows may experience the greatest change, as they shift from filtering characters to simply responding to virtual-key messages. This practice is demonstrated in the *Editor* demo discussed in this chapter.

Required Key Codes

Customarily, the WM_KEYDOWN message is trapped while looking for virtual key messages, each of which is identified by a constant with the format VK_XXXXXX, defined in WinUser.H.

Table S2.1 lists the virtual-key codes you'll encounter when programming for Windows keyboard events.

TABLE S2.1: Virtual Key Codes Defined in WinUser.H

Constant	Hex	Dec	Req	Keyboard	Comments
VK_LBUTTON	01h	1			Mouse emulation
VK_R1BUTTON	02h	2			Mouse emulation
VK_CANCEL	03h	3	✓	Ctrl+Break	Same as Ctrl+C
VK_MBUTTON	04h	4			Mouse emulation
	05 .. 07h				Not assigned
VK_BACK	08h	8	✓	Backspace	
VK_TAB	09h	9	✓	Tab key	
	0Ah .. 0Bh				Not assigned
VK_CLEAR	0Ch	12		Keypad 5	NumLock off
VK_RETURN	0Dh	13	✓	Enter	
	0Eh .. 0Fh				Not assigned
VK_SHIFT	10h	16	✓	Shift	Right or left
VK_CONTROL	11h	17	✓	Ctrl	Right or left
VK_MENU	12h	18	✓	Alt	Right or left
VK_PAUSE	13h	19		Pause	
VK_CAPITAL	14h	20	✓	Caps Lock	
	15h .. 19h				Reserved for Kanji system
	1Ah				Not assigned
VK_ESCAPE	1Bh	27	✓	Esc	
	1Ch .. 1Fh				Reserved for Kanji system

Continued on next page

TABLE S2.1 (CONTINUED): Virtual Key Codes Defined in WinUser.H

Constant	Hex	Dec	Req	Keyboard	Comments
VK_SPACE	20h	32	✓	Spacebar	
VK_PRIOR	21h	33	✓	Page Up	
VK_NEXT	22h	34	✓	Page Down	
VK_END	23h	35		End	
VK_HOME	24h	36	✓	Home	
VK_LEFT	25h	37	✓	Left Arrow	
VK_UP	26h	38	✓	Up Arrow	
VK_RIGHT	27h	39	✓	Right Arrow	
VK_DOWN	28h	40	✓	Down Arrow	
VK_SELECT	29h	41		Select	OEM specific
VK_PRINT	2Ah	42		Print	OEM specific
VK_EXECUTE	2Bh	43		Execute	OEM specific
VK_SNAPSHOT	2Ch	44		Print Screen	Win3 or later
VK_INSERT	2Dh	45	✓	Insert	
VK_DELETE	2Eh	46	✓	Delete	
VK_HELP	2Fh	47		Help	OEM specific
VK_0	30h	48	✓	0 through 9 keys	Same as ASCII 0 through 9 on main keyboard
..	..	57	✓		
VK_9	39h		✓		
	3Ah				Not assigned
	..				
	40h				
VK_A	41h	65	✓	A..Z, a..z keys	Main keyboard
..	✓		
VK_Z	5Ah	90	✓		

In Tables S2.1, S2.2 and S2.3, the column labeled Req (for “Required”) indicates by checks (✓) the keys that are required for all Windows implementations and thus will always be available. Also, some of the notations in the Comments column of these tables require a bit of explanation:

- Three values with VK_XXXX constants are identified as “mouse emulation.” This is not because they will be returned by the mouse as keyboard-event messages, but because these codes are sometimes used to emulate mouse-button events.
- Values that do not have VK_XXXX constants defined are noted as “not assigned.” These are not used and/or supported by any keyboard variations, nor are these assigned for any emulation purposes.
- Two groups in Table S2.1 are noted as “reserved for Kanji system.” These values are used with keyboards supporting the Japanese Kanji alphabet.
- Values noted as “OEM specific” may be supported by some keyboard variations but are not standard and should not be relied on except in special circumstances.

The virtual-key definitions do not include punctuation and symbols, and they do not distinguish between uppercase and lowercase. Also, applications should not attempt to use virtual-key definitions for text input.

New Windows Keys

The newer “Windows keyboards” include either two or three new keys. These may include two Windows keys, right and left, which call the Start menu, and a single Aps key. The function of the Aps key differs according to the active application, but, in general, it calls a pop-up menu in the same fashion as clicking the right mouse button does. These three new keys are defined in Table S2.2.

TABLE S2.2: Virtual Key Codes for New Windows Keys

Constant	Hex	Dec	Req	Keyboard	Comments
VK_LWIN	5Bh	91		Left Windows key	Not on all keyboards
VK_RWIN	5Ch	92		Right Windows key	Omitted on some portables
VK_APS	5Dh	93		Aps key	Not on all keyboards

NOTE

Because of space limitations, many portable computers provide only the left or right Windows key, not both.

Function Key and Other Special Key Codes

The function keys, keypad, arrow, Alt, Ctrl, and other special keys common to all keyboards are defined as shown in Table S2.3. In this table, along with some of the notations in the Comments column explained in the “Required Key Codes” section, those noted as “enhanced” are supported only by enhanced 101–102 keyboards.

Note that the left and right Shift, Ctrl, and Alt virtual keys are used only as parameters to the `GetAsyncKeyState` and `GetKeyState` functions. No other API functions or messages distinguish between the left and right keys in this fashion.

TABLE S2.3: Virtual Key Codes for Function Keys and Other Special Keys

Constant	Hex	Dec	Req	Keyboard	Comments
	5Eh				Not assigned
	..				
	5Fh				
VK_NUMPAD0	60h	96		Keypad 0	NumLock on
..	
VK_NUMPAD9*	69h	105		Keypad 9	
VK_MULTIPLY	6Ah	106		Keypad *	Enhanced
VK_ADD	6Bh	107		Keypad +	Enhanced
VK_SEPARATOR	6Ch	108			OEM specific
VK_SUBTRACT	6Dh	109		Keypad -	Enhanced
VK_DECIMAL	6Eh	110		Keypad .	Enhanced
VK_DIVIDE	6Fh	111		Keypad /	Enhanced
VK_F1	70h	112	✓	Function key F1	Standard
..	✓	..	
VK_F10	79h	121	✓	Function key F10	

Continued on next page

TABLE S2.3 (CONTINUED): Virtual Key Codes for Function Keys and Other Special Keys

Constant	Hex	Dec	Req	Keyboard	Comments
VK_F11	7Ah	122		Function key F11	Enhanced
VK_F12	7Bh	123		Function key F12	Enhanced
VK_F13	7Ch..	124		Function key F13	OEM specific
..	
VK_F24	87h	135		Function key F24	
	88h				Not assigned
	..				
	8Fh				
VK_NUMLOCK	90h	144	✓	NumLock	
VK_SCROLL	91h	145	✓	Scroll Lock	
	92h				Not assigned
	..				
	9Fh				
VK_LSHIFT	A0h	160		Left Shift key	Enhanced
VK_RSHIFT	A1h	161		Right Shift key	Enhanced
VK_LCONTROL	A2h	162		Left Ctrl key	Enhanced
VK_RCONTROL	A3h	163		Right Ctrl key	Enhanced
VK_LMENU	A4h	164		Left Alt key	Enhanced
VK_RMENU	A5h	165		Right Alt key	Enhanced
	A6h				Not assigned
	..				
	E4h				
VK_PROCESSKEY**	E5h	229			OEM specific
	E6h				Not assigned
	..				
	F5h				
VK_ATTN**	F6h	246			OEM specific
VK_CRSEL**	F7h	247			OEM specific
VK_EXSEL**	F8h	248			OEM specific

Continued on next page

TABLE S2.3 (CONTINUED): Virtual Key Codes for Function Keys and Other Special Keys

Constant	Hex	Dec	Req	Keyboard	Comments
VK_EREOF**	F9h	249			OEM specific
VK_PLAY**	FAh	250			OEM specific
VK_ZOOM**	FBh	251			OEM specific
VK_NONAME**	FBh	252			OEM specific
VK_PA1**	FDh	253			OEM specific
VK_OEM_CLEAR**	FEh	254			OEM specific
VK_KANA***	15h	21			Used for Kanji

*Because the keypad 5 key does not have a 'keypad' function, or at least not one that is commonly supported, the keypad 5 returns two different scan codes depending on whether the NumLock is set or cleared. See VK_CLEAR (0x0C).

**These VK_xxx definitions are found in the WinUser.H header file, but many of these are duplicated in the WinRes.H header.

***VK_KANA is defined in AfxVer.H.

The Editor Demo: Basic Keyboard Handling



The *Editor* demo demonstrates basic keyboard-event handling, including responses to keyboard messages and text-input handling. As you'll see, as word processors go, the *Editor* demo is both a wimp and an idiot. It includes provisions for nothing except the simplest operations for input, control, and for display. As it stands, *Editor* does not even toggle between insert and overwrite modes, normally a feature of even the dumbest editor.

Although it's severely lacking, the *Editor* program does accomplish its objective—not to build a word processor, but to show how keyboard-event messages are handled and to demonstrate the basic caret (text cursor) functions (discussed later in this chapter). For now, we will focus on how the *Editor* demo handles virtual-key code trapping.

NOTE

MFC-based applications may prefer to use the CEdit or CEditView classes to implement simple text-editor controls or windows without the labor of creating a full-featured editor.

In the following fragment from the *Editor* demo, the Home, End, Page Up and Page Down (VK_PRIOR and VK_NEXT), and arrow keys are trapped as virtual-key codes in the low word of the wParam argument that accompanies the WM_KEYDOWN message. Each key code exercises the appropriate control over the cursor position:

```
case WM_KEYDOWN:
    switch( LOWORD( wParam ) )
    {
        case VK_HOME:
            xCaret = 0;
            break;

        case VK_END:
            xCaret = cxBuff - 1;
            break;

        case VK_PRIOR:
            yCaret = 0;
            break;

        case VK_NEXT:
            yCaret = cyBuff - 1;
            break;

        ...

        case VK_DOWN:
            yCaret = min( yCaret+1, cyBuff-1 );
            break;
```

Since the *Editor* demo does not scroll and is limited to the client window display, the Page Up (VK_PRIOR) and Page Down (VK_NEXT) keys move the cursor to the top and bottom text positions within the window, respectively.

In other applications, almost any key that can be used to control, select, or activate can be found in the virtual-key list and trapped in a fashion similar to the one shown in the preceding example.

NOTE

The *Editor* demo is included on the CD that accompanies this book, in the Supplement 2 folder.

Getting Shift-State Data for Virtual Keys

The `wParam` and `lParam` arguments accompanying key-event messages carry a considerable amount of information, but they do not provide specific shift-key data other than for alphabetical characters (uppercase or lowercase) and other dual-character keys. To query the current shift states of the Shift, Ctrl, and Alt keys as well as the toggled shift keys, CapsLock and NumLock, you can use the `GetKeyState` function.

Realize, however, that the shift states reported by this function are the shift states associated with specific keyboard-event messages in the application's message queue—not the physical interrupt level state at the instant the inquiry is made. Thus, if the string “This is A TEST” were in the application's message buffer and the next char message to be read was a capital letter (assuming the right or left Shift key was used rather than the CapsLock key), the API function call `GetKeyState(VK_SHIFT)` would report the Shift key as down, regardless of the actual physical state of either Shift key at that instant. In like fashion, if the next char message were lowercase, the same inquiry would report both Shift keys were released.

For immediate information about the shift state of any of the shift keys—Shift, Ctrl, or Alt—use the `GetAsyncKeyState` function with the `VK_SHIFT`, `VK_CONTROL`, or `VK_MENU` parameter, respectively. Or, for even more specific information, you could use the `VK_Lxxxxx` or `VK_Rxxxxx` parameter to distinguish between the right and left Shift, Ctrl, or Alt key.

NOTE

You can also use the `VK_LBUTTON`, `VK_RBUTTON`, and `VK_MBUTTON` parameters with the `GetKeyState` function to query the mouse-button status. This is generally unnecessary, however, because the mouse-event messages already contain all the relevant information. See Supplement 3 for more information about mouse-event messages.

Handling Text Input

You already know how important the keyboard is as an input device. What is even more important is the ability of an application to cleanly receive and display text—anything from a few lines to full screens of structured or unstructured text.

And, in any case, there are a few basics that apply to all types of text input. Of course, with Windows, there are a few differences as well. As the Walrus said, “the time has come to speak of many things ...,” but specifically, to speak of things governing Windows 98’s ability to display text and to accept inputs.

The Caret versus the Cursor

In Windows, the word *cursor* has been reserved for the mouse cursor. The familiar DOS cursor (from the text-mode display) is now renamed the *caret*. Technically, this term more properly refers to the curious little hat-shaped character (^), which C/C++ commonly recognizes as the bitwise XOR operator, and which other human European languages use as an accent (as in â, for example).

Of course, the caret displayed has no more resemblance to the caret character (thus far, no applications have appeared using a literal caret), than the cursor (mouse) resembles the DOS text cursor. In both cases, the displayed symbol for either the cursor or caret pointer device is a graphic and therefore flexible. Ergo, applications are free to modify both devices, although some standards and conventions do apply.

One standard that does not apply in Windows, however, is the DOS cursor standard of a blinking underbar cursor (caret). This is because Windows supports both flexible font sizes and proportionally spaced fonts, so no fixed character width applies, and the underline caret simply does not serve as an accurate position indicator. Instead, a blinking vertical line, the same height as either the font or the interline spacing, has replaced the blinking underbar in a wide variety of applications. Conventionally, the blinking line is positioned at the point where the next character will begin (or at the first or leftmost extent of an existing character).

For languages written from right to left, such as Hebrew, other conventions apply. For a vertical script, such as ancient Chinese ideograms, the solution might be a blinking horizontal bar. Of course, if a “bostriphon” (literally, as the ox plows) script remains in use anywhere in the world, the alternating right-to-left and left-to-right text would demand its own standards. In fact, this might be a situation where, curiously enough, the caret character positioned below the line, might serve nicely—certainly much better than a plain underbar.

Changing the Caret Type and Position

Because the caret (text cursor) is a system resource, individual applications are not free to create or destroy the caret any more than they are free to create and destroy the mouse cursor. Applications may, however, borrow the system caret—as long as the application holds the system (input) focus. During this time, an application can change the caret type, as well as control the caret position.

Caret Focus Messages

The first step, before the caret can be positioned or modified, is for the application to know when it gains or loses the system focus. Two Windows messages are devoted to precisely this function:

- `WM_SETFOCUS` notifies an application that it is receiving the system focus.
- `WM_KILLFOCUS` notifies an application that it is losing the system focus.

The two focus messages are always issued in pairs; that is, a `WM_SETFOCUS` will always be followed at some point by a `WM_KILLFOCUS` message, while the latter message will never be issued except when preceded by the former.

NOTE

`WM_SETFOCUS` and `WM_KILLFOCUS` are totally independent of the `WM_PAINT`, `WM_SIZE`, and other messages that instruct applications to update or adjust their displays. Applications can accomplish these and other appropriate tasks without receiving the system focus or coming to the front of the screen.

At the same time, receipt of a `WM_SETFOCUS` or `WM_KILLFOCUS` message does not indicate or suggest that an application is being created or destroyed. It just indicates that the focus is being shifted to or from the present application.

Conversely, a `WM_CREATE` message is always preceded by a `WM_SETFOCUS` message. In parallel fashion, a `WM_KILLFOCUS` message follows a `WM_DESTROY` message, providing opportunities to create and destroy application carets. Depending on your application's needs, you may provide responses to these event messages or simply ignore them as irrelevant.

Caret Shape and Position Functions

When an application receives the system focus, as notified by a `WM_SETFOCUS` message, the immediate response, if applicable, is usually to call the `CreateCaret` function to assign the desired caret shape, followed by `SetCaretPos` to position the caret and `ShowCaret` to make the caret visible:

```
case WM_SETFOCUS:
    CreateCaret( hwnd, (HBITMAP) 1, cxChr, cyChr );
    SetCaretPos( xCaret * cxChr, yCaret * cyChr );
    ShowCaret( hwnd );
    break;
```

The `CreateCaret` function is called with four parameters: a handle identifying the application window owning the caret and a bitmap handle providing the caret shape, and the width and height of the caret. The bitmap handle may have two default values: `NULL` to create a solid caret or one (1) to create a gray caret. In both cases, the caret will be a block (or line) with the dimensions specified by the third and fourth arguments. In the *Editor* demo, the caret is a gray block that is the height and width of a single character.

Because any call to `CreateCaret` destroys any previous caret shape, applications must be prepared to re-create their caret shape anytime the system focus is received. If this is not done, the caret displayed will be whatever shape was set by the last application holding the system focus and defining a caret format.

In addition to creating a caret (text cursor), functions are also provided to set the caret position, find the caret position, and make the caret visible.

The `SetCaretPos` function is called with the x-axis and y-axis coordinates for the caret position, nominally the position of the upper-left corner of the caret bitmap. If, however, the window was created using the `CS_OWNDC` class style, these coordinates are mapped to the mapping mode associated with the window, and the caret position—and appearance—are affected accordingly. (Mapping modes will be discussed and illustrated in Chapter 23.) The caret position is set regardless of whether the caret is visible.

The complementary function, `GetCaretPos`, is called with a long pointer to a `POINT` structure. It returns the caret's current position in client window coordinates or returns `FALSE` on failure.

The `ShowCaret` function is called to make the caret visible and has only one parameter: the handle identifying the window owning the caret. If the caret has

been hidden two or more times in succession, the `ShowCaret` function must be called an equal number of times before the caret will become visible. Also, if the handle passed is `NULL`, the `SetCaret` function will work only if the caret is owned by a window in the current task (it is, however, very bad form for one application to try to show another application's caret).

NOTE

If the caret has no shape or size, the `ShowCaret` function will have no effect.

The flip side of the `WM_SETFOCUS` message is the `WM_KILLFOCUS` message, which includes the `HideCaret` function.

```
case WM_KILLFOCUS:
    HideCaret( hwnd );
    DestroyCaret();
    break;
```

The `HideCaret` function, like its counterpart `ShowCaret`, is called with a single parameter identifying the application window owning the caret. Hiding the caret does not destroy the caret shape, which can be restored using the `ShowCaret` function. Multiple successive calls to the `ShowCaret` function must be matched by multiple calls to the `HideCaret` function before the caret will be hidden.

Also, if desired, the `SetCaretBlinkTime` function sets the caret blink rate as elapsed milliseconds between flashes. The function is called with a single argument (`UINT`), specifying both the delay between flashes and the flash duration.

The `GetCaretBlinkTime` function requires no parameters and returns a `UINT` value, specifying the blink time in milliseconds.

Caret (Cursor) Positioning for a Fixed-Width Font

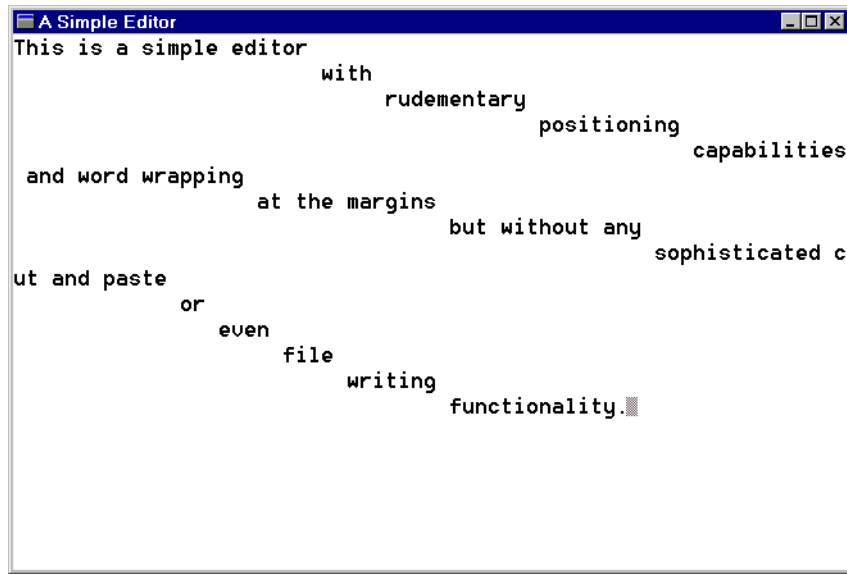
The *Editor* demo uses a gray-block caret (see Figure S2.3) sized to fit the `SYSTEM_FIXED_FONT`. Because this is a fixed-width font, cursor positioning is quite simple.

With the caret initially located at the upper-left corner of the client window, the caret positioning provisions begin by watching for a `WM_KEYDOWN` message, such as:

```
case WM_KEYDOWN:
    switch( LOWORD( wParam ) )
    {
```

FIGURE S2.3:

A simple text-only editor program with a gray-block caret



Most of the `WM_KEYDOWN` messages received will be ignored; a response will be generated only when the `wParam` argument identifies one of the cursor or page keys (identified by the appropriate `VK_xxxx` virtual key messages, as explained earlier in the chapter). Because this is a fixed-width font, the appropriate response is simply to increment or decrement the cursor in character (row/column) positions.

```
case VK_HOME:
    xCaret = 0;
    break;

case VK_END:
    xCaret = cxBuff - 1;
    break;

case VK_PRIOR:
    yCaret = 0;
    break;

case VK_NEXT:
    yCaret = cyBuff - 1;
    break;
```

```
case VK_LEFT:
    xCaret = max( xCaret-1, 0 );
    break;

case VK_RIGHT:
    xCaret = min( xCaret+1, cxBuff-1 );
    break;

case VK_UP:
    yCaret = max( yCaret-1, 0 );
    break;

case VK_DOWN:
    yCaret = min( yCaret+1, cyBuff-1 );
    break;
```

NOTE

For a variable-width font, such as the default font used for a WYSIWYG (“What You See Is What You Get”) editor, the immediate response in most cases would be essentially the same. The application would keep track of the cursor position in terms of line and character positions and only convert these row/column equivalents to an actual screen position matching the string display position immediately before calling the `SetCaretPos` function.

Deleting the Character at the Caret Position

The page and arrow keys are only a few of the `VK_XXXX` messages to which the *Editor* demo could respond. In this example, there’s only one additional virtual key provided for: the `VK_DELETE` event, which is handled as:

```
case VK_DELETE:
    for( x = xCaret; x < cxBuff - 1; x++ )
        Buffer( x, yCaret ) = Buffer( x+1, yCaret );
    Buffer( cxBuff-1, yCaret ) = ' ';
```

Here, the response is a bit more elaborate than simply changing the cursor position. In this instance, the program deletes the character at the cursor position by shifting the remainder of the line left one character position and appending a blank at the end of the line. (The space ensures that the character previously following is overwritten with a blank.)

Updating the text buffer is only a part of the necessary response. The display also needs to be updated, which could be done in a couple of ways. For one, the application could simply invalidate the appropriate region, allowing a repaint to repair the screen. Or, as done in this case, an immediate screen update could be executed.

```
HideCaret( hwnd );  
hdc = GetDC( hwnd );  
SelectObject( hdc,  
              GetStockObject( SYSTEM_FIXED_FONT ) );  
TextOut( hdc, xCaret * cxChr, yCaret * cyChr,  
         &Buffer( xCaret, yCaret ), cxBuff-xCaret );  
ShowCaret( hwnd );  
ReleaseDC( hwnd, hdc );
```

Notice that before updating the screen, the `HideCaret` function is called to remove the caret from the display, and after updating, `ShowCaret` restores the text cursor with the position unchanged. Removing the caret is a necessary operation anytime the screen is repainted, simply to ensure that the caret doesn't interfere with the paint operation (similar precautions are generally used when the mouse cursor is active). Feel free to experiment by commenting out both the `HideCaret` and `ShowCaret` API calls and observing the results directly.

For a `VK_BACK` (backspace) message, essentially the same response could be used, except that the program would also need to decrement the caret position. An alternative is to handle this through a `WM_CHAR` message, as explained shortly, in the “Handling `WM_CHAR` Messages” section.

Other possible `VK_XXXXX` messages might require quite different responses. For example, suppose provisions were made for the `VK_INSERT` message to toggle between insert and overwrite modes. Should the cursor shape, size, or format change to reflect the current mode?

Still, whatever operations are provided, the last provision within the `VK_KEYDOWN` response is to update the caret position, even though many of the options may not have affected this position at all.

```
SetCaretPos( xCaret * cxChr, yCaret * cyChr );  
break;
```

Calculating the position is simple using a fixed-width font; it requires nothing more than multiplying the row and column position by the character width and line spacing, respectively.

A different approach is required for a variable-width font. There are several possibilities, but perhaps the best approach might lie in the *current position* or *cp*, a Windows feature described in the next section.

Handling Carets for Variable-Width Fonts

The *cp* is an internal `POINT` structure that can be used by Windows to track drawing operations, with a separate *cp* for each device context. In some cases, *cp* is ignored and not updated during drawing operations.

For example, the `TextOut` function used to write (draw) the text display in the *Editor* demo does not normally keep track of the *cp*. However, this is subject to change. By calling the `SetTextAlign` function with the `fmode` argument, `TA_UPDATECP` will enable current position tracking.

When current position tracking is enabled, the `GetCurrentPositionEx` function can be called to retrieve the *cp* coordinates after writing a string or a portion of a string. For example, assume that the string displayed reads, “This is a positioning text,” and the caret should be positioned immediately after the *a*. With a proportionally spaced font, the capital *T* will be wider than average, and the two *i*’s will be narrower. Obviously, attempting to estimate character positions from the `tmAveCharWidth` spacing is not going to produce accurate results. If, however, only a part of the string is drawn (“This is a”), and *cp* is retrieved before completing the sentence, the retrieved *cp* will provide the positioning for the caret.

The bad news is that it’s not quite as simple as this illustration suggests. You will need to refer to the `SetTextAlign`, `GetCurrentPositionEx`, and `TextOut` functions for details on how to use each appropriately for the task. And, of course, you will need to do a bit of experimenting.

Handling WM_CHAR Messages

Like `WM_KEYDOWN` messages, `WM_CHAR` messages are also subject to a wide variety of processing. Most of the special provisions discussed here can also be handled by virtual-key responses in the `WM_KEYDOWN` message handling. In fact, they would be handled that way in most cases. But the `WM_CHAR` message-handling methods are alternatives that you may want to consider.

TIP

For an exercise in keycodes and functions, convert as much as is practical of the WM_CHAR message handling to WM_KEYDOWN handling. Just be sure to test your results carefully.

Repeat Characters

The first step, because the WM_CHAR message may well include a repeat count, is a loop controlled by the low word in the lParam argument:

```
case WM_CHAR:
    for( i = 0; i < (int)LOWORD(lParam); i++ )
    {
```

Within the loop, even though the high word of wParam should be simply a NULL, the LOWORD macro is used to discard any potentially conflicting data:

```
        switch( LOWORD( wParam ) )
        {
```

Backspace

The first char value trapped is the backspace character (\b).

```
case '\b':           // backspace
    if( xCaret > 0 )
    { xCaret--;
      SendMessage( hwnd, WM_KEYDOWN, VK_DELETE, 1L );
    }
    break;
```

The Backspace key is easily handled by simply decrementing the caret position and then issuing a key down/delete key message. Alternatively, this could be handled as a virtual key (VK_BACK) in the preceding WM_KEYDOWN handler. In this case, the handling might well be almost exactly the same as shown here.

Tab

The tab char is easily provided for using a do .. while loop to insert spaces repeatedly until the desired character position is reached.

```

case '\t':           // tab
do
    SendMessage( hwnd, WM_CHAR, ' ', 1L);
while( xCaret % 8 != 0 );
break;

```

Like the Backspace key, the Tab key can also be trapped by the preceding WM_KEYDOWN handler.

Carriage Return and Line Feed

The next two char events are handled as a pair. The first character watched for is the carriage return (\r or ASCII 0x0D), which resets the horizontal position to the beginning of the line. It is then allowed to fall through to the second case, the line feed, for further response.

```

case '\r':           // carriage return
    xCaret = 0;       // falls through to '\n'

case '\n':           // line feed
    if( ++yCaret == cyBuff )
        yCaret = 0;
    break;

```

The line feed character (\n or ASCII 0x0A), by convention, does not reset the horizontal position; instead, it is treated as the equivalent of the down arrow. In this example, both of these responses could as easily have been written as:

```

case '\r':
    SendMessage( hwnd, WM_KEYDOWN, VK_HOME, 1L );

case '\n':
    SendMessage( hwnd, WM_KEYDOWN, VK_DOWN, 1L );
    break;

```

Notice that the carriage return response is still allowed to fall through to the subsequent line feed response.

While the practice of responding to one keyboard-event message by issuing other keyboard-event messages may, at first, seem slightly redundant, the overall result is an economy of effort both for the programmer and for the program. After all, instead of duplicating essentially the same response (both as code and executable), this approach allows code and executable to do double duty. In addition, this approach can be (and often is) applied to features other than keyboard responses.

Escape (Esc)

The Escape key (ASCII 0x1B) is another popular *hotkey*. In this example, it resets the text buffer and then issues a query for confirmation:

```
case '\x1B':           // escape
    if( MessageBox( hwnd, "Reset text buffer?",
        "Editor Query",
        MB_ICONEXCLAMATION | MB_OKCANCEL |
        MB_DEFBUTTON2 ) == IDOK )
```

The `MessageBox` API call presents a stock dialog box with the caption “Editor Query,” the message “Reset test buffer?” and the OK and Cancel buttons. If the OK button is clicked, the function returns `TRUE`; if the Cancel button is clicked, it returns `FALSE`. This value dictates whether or not the following provisions will be executed.

If the decision is to proceed, then a double loop overwrites the text buffer with blanks, the caret position is reset to the first character position at the upper-left, and last, the `InvalidateRect` function is called to clear the existing display by issuing a `WM_PAINT` message.

```
{
    for( y = 0; y < cyBuff; y++ )
        for( x = 0; x < cxBuff; x++ )
            Buffer( x, y ) = ' ';
    xCaret = 0;
    yCaret = 0;
    InvalidateRect( hwnd, NULL, FALSE );
}
break;
```

Other Character Events

As a final response to `WM_CHAR` messages, the default provision handles all other character events, which are assumed to be conventional alphabetic, numeric, or punctuation characters:

```
default:           // all other chars
    Buffer( xCaret, yCaret ) = (char) LOWORD( wParam );
    HideCaret( hwnd );
    hdc = GetDC( hwnd );
    SelectObject( hdc,
        GetStockObject( SYSTEM_FIXED_FONT ));
```



```

        TextOut( hdc, xCaret * cxChr, yCaret * cyChr,
                &Buffer( xCaret, yCaret ), 1 );
        ShowCaret( hwnd );
        ReleaseDC( hwnd, hdc );
        if( ++xCaret == cxBuff )
        {
            xCaret = 0;
            if( ++yCaret == cyBuff ) yCaret = 0;
        }
        break;
    } }

```

The handling used is essentially the same as shown earlier for the `VK_DELETE` message. However, in view of earlier remarks about sharing responses, couldn't the present duplication of code and executable be similarly avoided?

Finally, because some of the preceding responses have affected the caret position, the same closing provision is required here as in the `WM_KEYDOWN` response.

```

        SetCaretPos( xCaret * cxChr, yCaret * cyChr );
        break;

```

Generating Event Messages

The flip side of processing keyboard messages (or any other event messages) is being able to generate your own messages to request specific actions. You've seen several brief examples of message generation in the preceding code fragments.

Sending Messages to Applications

The `SendMessage` function is called with the same four parameters that are passed to the `WndProc` procedure:

```

SendMessage( HWND hwnd,    UINT msg,
             DWORD wParam, LONG lParam );

```

`SendMessage` passes its arguments to Windows, which then places the message in the message queue for the application identified by the `hwnd` argument. In this fashion, the destination could be the same application window that originated the message, another window belonging to the same application, or even a window belonging to another application entirely.

PostMessage versus *SendMessage*

Two functions are provided for passing messages within an application: **SendMessage** and **PostMessage**. These work essentially the same way except for how they dispatch messages.

PostMessage places the message in the messaging queue and then returns immediately—without waiting for the message to be delivered. Using **PostMessage**, the posting procedure can continue operating and the called procedure—the message recipient—does not act until the message queue delivers.

In contrast, **SendMessage** places a message in the queue but does not return until the message has been processed and delivered to the recipient. In effect, **SendMessage** transfers operational control to another routine—the message recipient—and waits for the recipient to finish its task and return control. For sending messages to other applications, **SendMessage** is not the most efficient method.

Scrolling with Arrow Keys



The *PainText* demo (discussed in Supplement 1) demonstrates how scrollbars are used to respond to mouse-event messages. Even though it is rare to find a computer without a mouse (at least, one that is running Windows), there may be times when using the keyboard for scrolling is more convenient.

To further demonstrate the **SendMessage** function, here is a patch for the *PainText* program, which allows the arrow keys to simulate mouse operations:

```
switch( msg )
{
    ...
    case WM_KEYDOWN:
        switch( LOWORD( wParam ) )
        {
            case VK_LEFT:
                SendMessage( hwnd, WM_HSCROLL, SB_LINEUP, 0L );
                break;
            case VK_RIGHT:
                SendMessage( hwnd, WM_HSCROLL, SB_LINEDOWN, 0L );
                break;
```

```
        case VK_UP:
            SendMessage( hwnd, WM_VSCROLL, SB_LINEUP, 0L );
            break;

        case VK_DOWN:
            SendMessage( hwnd, WM_VSCROLL, SB_LINEDOWN, 0L );
            break;

        ...
    }
    break;

    ...
}
break;

...
```

In this fashion, the four arrow keys use the `SendMessage` function to generate scrollbar messages equivalent to clicking on the arrow keys at the ends of the scrollbars. Alternatively, for faster scroll operations, the Page Up (`VK_PRIOR`), Page Down (`VK_NEXT`), Home (`VK_HOME`), and End (`VK_END`) keys can be used to send the appropriate `SB_PAGEUP` and `SB_PAGEDOWN` messages to each scrollbar.

The `SendMessage` function can be used to generate any valid Windows messages, not just those shown here. Furthermore, this can be initiated in response to any appropriate circumstance, not just a keyboard (or mouse) event.

In Windows programming, messages are generated in response to a wide variety of circumstances and for a wide variety of purposes. In many senses, the message functions and operations are the heart of Windows applications (actually, life's blood might be a better metaphor).

In this chapter, we covered handling keyboard-event messages. In the next chapter, you will learn about handling mouse-event messages.

S U P P L E M E N T

T H R E E

S3

Mouse Operations

- Mouse-event messages
- Mouse-movement tracking
- Mouse cursor shapes
- Mouse-hit testing

If you have read the book up to this point or have had time to work on a Windows system (and who hasn't?), it should be quite obvious that Windows is totally mouse-oriented. Granted, there are keyboard options to permit switching between windows and applications without using the mouse but in reality, without a mouse, using Windows is almost impossible. At the same time, it is possible that your application could be designed in such a fashion that a mouse was not required—possible, but unlikely.

Besides, is there any real reason—short of some mechao-myomorphicphobia—for seeking to avoid these ubiquitous little pseudo-rodents? (Obviously, the question demands a negative response.)

NOTE

If you will pardon the bad Greek/Latin, *mechao-myomorphicphobia* is a fear of mechanical mice.

In any case, hypothetical phobias aside, at least a minimal knowledge of mouse operations is essential to any application. Furthermore, knowing how the mouse works may even suggest uses and possibilities relevant to your application design.

The Evolution of the Genus MusMechano

Originally, mouse devices were single-button (left-button equivalent) pointing devices—a primitive form that is still found today on Apple/Macintosh systems but that is virtually obsolete on contemporary DOS, Windows, and OS/2 systems (despite the fact that some sources continue to suggest that a single-button mouse should be considered a minimal standard).

For Windows 2000, two standard mouse configurations are supported: the two-button variety typified by the standard Microsoft Mouse and the three-button variety represented by the Logitech Mouse. Support is also provided for some variant forms, such as mouse-emulation by joy sticks and lightpens, which are treated as single-button mice. A variety of less common devices—such as those for use by disabled persons or special-purpose variations (“un-mouse” pads, touch pads, joy-stick pointers, and so on)—attempt to make their interfaces indistinguishable from one of the standard interfaces.

Some mouse varieties (“mutations,” if you prefer) have appeared, sporting dozens of “keys.” These keys are usually used for numerical data operations, but in theory, they provide mini-keyboards on a mouse. These varieties require special drivers before any Windows 98 interface is possible, and for now, they can simply be ignored. Virtual devices, such as control gloves that sense spatial motion, are still experimental, and they also can be ignored for now.

Also, the newest “mouse” from Microsoft (and others) includes a wheel between the two buttons. The wheel is designed as an add-in for scrolling through Web pages but—at present—should not be considered a “standard” requiring support. For the moment, only the two “standard” mouse types require consideration. Even though the middle button on three-button mice can be extremely useful, this chapter concentrates on the minimalist standard of two mouse buttons: left and right.

Common usage emphasizes the left mouse button, regardless of the actual number of buttons available, as the equivalent of the Enter key. The right button is often used as the equivalent of the Escape key (south-paws can reverse this by going to the Control Panel, selecting the Mouse icon, and swapping the left and right buttons). When available, the middle button triggers optional shortcuts. If you are using a Logitech or equivalent mouse—one with a third button—the third button is still active, even if none of your applications respond to the middle-button-down (`WM_MBUTTONDOWN`) messages.

Is There a Mouse in the House?

Although it is usually safe to assume that a mouse is present, for critical applications, it is possible to query the system to ensure that a mouse is present. This task is accomplished using the `GetSystemMetrics` function:

```
if( GetSystemMetrics( SM_MOUSEPRESENT ) ) ...
```

If a mouse is present (and working), `GetSystemMetrics` will return `TRUE`; if no mouse is available, `GetSystemMetrics` will report `FALSE`. If the result is `FALSE`, mouse-critical applications can report accordingly.

How an application should respond to the absence of a mouse depends entirely on the application and the importance of mouse support. One possibility is to abort the program execution, as demonstrated in the *Mouse3* demo discussed later in this chapter.

Mouse Actions and Events

Three principal types of mouse actions are possible:

Clicking Pressing and releasing a mouse button

Double-clicking Clicking a mouse button twice rapidly

Dragging Moving the mouse while holding down a mouse button

Other mouse actions may be implemented by an application; these are generally more a feature of the application than a standard mouse activity. For example, instead of dragging an object such as an icon to move it, some applications permit you to simply click once to select it and click again to release it in its new position, without holding down the mouse button. This form (and variations) are popular with many drawing programs and help to reduce mouse-wrist injuries (muscle/tendon strains caused by holding and dragging the mouse in applications requiring fine control for positioning).

Mouse events in Windows are different from mouse events under DOS in several respects:

- In the Windows environment, mouse-button events are not always paired because the environment is shared. For example, a button-down event can occur in one window while the release event is not reported until the mouse has entered another window. In this fashion, the application receives only one of these event messages, and if it depends on receiving both, may malfunction. For an example, see the discussion of the *Mouse2* demo later in this chapter.
- An application in Windows can hold the mouse focus, even after relinquishing the system focus. The application can also continue to receive all mouse messages, even though the mouse is outside the client window area.
- If a system-modal message or dialog box is active in a Windows application, no other application window can receive any mouse messages. System modal messages and dialog boxes prohibit switching to any other window or application until they are exited.

Special circumstances aside, however, there are normally no restrictions or special provisions required for handling mouse-event messages in Windows.

Mouse-Event Messages

A total of 22 mouse-event messages are defined in `WinUser.H`. Two of these have values duplicating other mouse messages and are apparently intended for internal use (or, perhaps, simply for variety). Of the remaining 20, these messages occur in pairs: one for client window events (mouse events occurring within the application’s client window) and a corresponding event message for mouse actions that occur outside the client window. Table S3.1 lists these mouse-event messages and their values.

TABLE S3.1: Mouse-Event Messages and Values

Client Window Events	Value	Nonclient Window Events	Value
WM_MOUSEMOVE*	0x0200	WM_NCMOUSEMOVE	0x00A0
WM_LBUTTONDOWN	0x0201	WM_NCLBUTTONDOWN	0x00A1
WM_LBUTTONUP	0x0202	WM_NCLBUTTONUP	0x00A2
WM_LBUTTONDOWNBLCLK	0x0203	WM_NCLBUTTONDOWNBLCLK	0x00A3
WM_RBUTTONDOWN	0x0204	WM_NCRBUTTONDOWN	0x00A4
WM_RBUTTONUP	0x0205	WM_NCRBUTTONUP	0x00A5
WM_RBUTTONDOWNBLCLK	0x0206	WM_NCRBUTTONDOWNBLCLK	0x00A6
WM_MBUTTONDOWN	0x0207	WM_NCMBUTTONDOWN	0x00A7
WM_MBUTTONUP	0x0208	WM_NCMBUTTONUP	0x00A8
WM_MBUTTONDOWNBLCLK**	0x0209	WM_NCMBUTTONDBLCLK	0x00A9

*WM_MOUSEFIRST duplicates WM_MOUSEMOVE.
**WM_MOUSELAST duplicates WM_MBUTTONDOWNBLCLK.

Normally, mouse-movement and mouse-button messages are reported only while the mouse remains within the application’s client window. In these cases, the nonclient window mouse events are not issued to the application. However, there are circumstances where an application will request and track mouse events outside its own immediate jurisdiction; for example, a screen-capture application needs this information.

NOTE

Because nonclient mouse messages are relevant only under special circumstances, they are not demonstrated in the examples in this chapter. They will be demonstrated in later chapters dealing with graphics, in Part 3 of this book.

The most important mouse-event messages for most applications are the nine client window mouse-button messages listed in Table S3.2.

TABLE S3.2: Mouse-Button Messages

Button	Pressed	Released	Double-Clicked
Left	WM_LBUTTONDOWN	WM_LBUTTONUP	WM_LBUTTONDOWNBLCLK
Right	WM_RBUTTONDOWN	WM_RBUTTONUP	WM_RBUTTONDOWNBLCLK
Middle	WM_MBUTTONDOWN	WM_MBUTTONUP	WM_MBUTTONDOWNBLCLK

WM_xBUTTONDOWN and WM_xBUTTONUP messages are issued only once—when a mouse button is pressed or released. A WM_xBUTTONDOWNBLCLK message is issued only when a mouse button is rapidly double-clicked (press+release+press). Unlike keyboard events, repeating mouse-button messages are not issued by holding down a mouse button, even though WM_MOUSEMOVE messages are issued for all mouse movements regardless of the button states.

The status of both (or all three) mouse buttons may, however, be retrieved from the information accompanying the WM_MOUSEMOVE message, as explained in the “Information in Mouse Messages” section, coming up soon.

Double-Click Messages

Because not all applications require (or desire) double-click event messages, provisions are included in Windows 98 to control whether or not double-click events are reported. These provisions are controlled by the client window’s (or child window’s) style definition. WM_xxxDBLCLK messages are generated only if the class style includes the CS_DBLCLKS flag, as in:

```
wc.style = CS_HREDRAW | CS_VREDRAW | CS_DBLCLKS;
```

If the CS_DBLCLKS flag is not set, a double-click is simply received as four separate events: WM_xBUTTONDOWN, WM_xBUTTONUP, WM_xBUTTONDOWN, and WM_xBUTTONUP. However, when CS_DBLCLKS is enabled, a double-click is received

as `WM_xBUTTONDOWN`, `WM_xBUTTONUP`, `WM_xBUTTONDBLCLK`, and `WM_xBUTTONUP`, with the double-click message replacing the second-button-pressed message.

In general, responses to a double-click message are designed as continuations or expansions of single-click responses; before the double-click event is registered, the application has already received a button-pressed/button-released event message pair.

Also, if single- and double-click events are intended to perform quite different tasks, the response processing could become quite complex because the single-click event message will always be received before the double-click event. On the other hand, you can make this work for you.

For example, consider the Windows File Manager's handling of a single-click and a double-click on a subdirectory listing. A single-click changes the active directory; a double-click calls a new directory window displaying the selected directory. The result is that accidental entries perform in very much the same fashion, if not precisely the same, as the intended result.

NOTE

Under Windows 98 (and 95), File Manager is still available; it's in the Windows directory under the filename `WinFile.EXE`. Windows Explorer works in a quite different fashion than the File Manager example cited here.

Mouse-Movement Messages

Although many applications require only mouse-button event messages, the `WM_MOUSEMOVE` message is issued every time the mouse moves physically. However, this does not necessarily mean that the mouse has moved on the screen because movement is reported not per screen pixel but per unit of mouse motion.

Rapid mouse movement may cause `WM_MOUSEMOVE` messages to be reported irregularly. The *Mouse1* demo, discussed later in this chapter, may demonstrate this effect. This effect is produced partially by the accelerator settings, which multiply distances for rapid mouse movements, and partially by the inability of the system to respond to rapid movement events.

With slower mouse movement, the `WM_MOUSEMOVE` messages will most likely overlap, resulting in more than one message with the same screen coordinates. Overlapping mouse coordinates, however, ensure a solid series of painted screen coordinates.

NOTE

Mouse movement is reported in *mickeys*. (You already know that all programmers are punsters.) A high-resolution mouse reports 200 to 300 mickeys per inch (without acceleration). The speed of movement is reported as mickeys per second. Given the predilection of programmers for puns, it seems a wonder that this API was not named `MouseTrap()`.

Miscellaneous Mouse Messages

Three additional mouse messages are possible, but normally, they are not the direct concern of the application itself and are left to Windows for appropriate handling.

WM_MOUSEACTIVATE Occurs when the cursor is in an inactive window and any mouse button is pressed.

WM_MOUSEENTER Occurs when the mouse enters any window.

WM_MOUSELEAVE Occurs when the mouse leaves any window.

Information in Mouse Messages

Each mouse-event message contains, in addition to the event itself, complete mouse button and Shift- and Ctrl-key status data in the `wParam` argument (as flag information), and mouse coordinate information in the `lParam` argument. The status data can be tested as:

```
if( wParam & MK_LBUTTON )...    // left button down
if( wParam & MK_RBUTTON )...    // right button down
if( wParam & MK_MBUTTON )...    // middle button down
if( wParam & MK_CONTROL )...    // Ctrl key pressed
if( wParam & MK_SHIFT )...      // Shift key pressed
```

The mouse coordinate information is in client-window pixel coordinates relative to the upper-left corner. As mentioned previously, this information is found in the `lParam` argument, with the x-coordinate in the low-order word and the y-coordinate in the high-order word. You can use the `MAKEPOINTW` macro to convert the `lParam` argument to a `POINTW` structure.

The *Mouse1* Demo: Tracking the Mouse



The *Mouse1* demo demonstrates how WM_MOUSEMOVE messages are tracked. It plots a single pixel at the coordinates reported with each message received. Plotting is toggled on and off by the left mouse button. Plotting begins with one WM_LBUTTONDOWN message and ends with the next WM_LBUTTONDOWN message. For this example, only the WM_MOUSEMOVE and WM_LBUTTONDOWN messages are provided with responses.

The WM_LBUTTONDOWN message is used to toggle the fPaint variable by XORing fPaint and 1, flipping the value from TRUE to FALSE and vice versa. The value of fPaint was initialized as zero (0).

```
case WM_LBUTTONDOWN:
    fPaint ^= 1;
    MessageBeep(0);
    break;
```

As a minor bonus, the MessageBeep function is called to provide audio feedback. You can call MessageBeep with a zero argument, although this argument is simply the equivalent of MB_OK and produces the system default sound. Other options are listed in Table S3.3.

TABLE S3.3: MessageBeep Arguments

Argument	Sound
0xFFFFFFFF	Standard beep using the computer speaker
MB_ICONASTERISK	SystemAsterisk
MB_ICONEXCLAMATION	SystemExclamation
MB_ICONHAND	SystemHand
MB_ICONQUESTION	SystemQuestion
MB_OK	SystemDefault

The second response to the `WM_MOUSEMOVE` message depends on the current value in `fPaint` (`TRUE` or `FALSE`). It extracts the mouse's window coordinates from `lParam` and paints a single black pixel.

```
case WM_MOUSEMOVE:
    if( fPaint )
    {
        hdc = GetDC( hwnd );
        SetPixel( hdc, LOWORD( lParam ), HIWORD( lParam ), 0L );
        ReleaseDC( hwnd, hdc );
    }
    break;
```

For tracking mouse movements, this method is sufficient. However, pay particular attention to how different rates of movement affect the dot spacing.

NOTE

The *Mouse1* demo is included on the CD that accompanies this book, in the Supplement 3 folder.

The Mouse Cursor

Supplement 2 explained how the text pointer is now known as the *caret*, and the term *cursor* is now the property of the mouse pointer. And, while the mouse pointer points at a single pixel rather than a character cell position, the cursor proper is a bitmapped image that is tracked across the display while preserving the background image. One pixel location within this cursor image is known as the *hot spot*, which is the actual pointer location tracked.

Windows 2000 provides 14 predefined cursor images, which are listed in Table S3.4. By default, Windows uses the slanted arrow (`IDC_ARROW`). Individual applications are free to define any of these standard cursors as their own default cursor. (Most of the applications discussed in this book follow Windows 2000 in using the slanted arrow default.)

TABLE S3.4: Windows Predefined Cursor Shapes

Cursor	Description
IDC_APPSTARTING	A standard arrow and small hourglass combination; used to show that an application is opening
IDC_ARROW	An arrow pointing diagonally up and left; the familiar default cursor
IDC_CROSS	A simple horizontal/vertical cross
IDC_IBEAM	An I-beam cursor; commonly used for word processing
IDC_ICON	An empty shape that can be used to hide the cursor; i.e., no cursor image (<i>obsolete</i>)
IDC_NO	An international negative; i.e., a circle with a diagonal slash
IDC_SIZE	<i>Obsolete</i> , see IDC_SIZEALL
IDC_SIZEALL	Four arrows, pointing in the four cardinal directions (up, down, left, and right)
IDC_SIZENESW	A tilted double-arrow, pointing diagonally up to the right (NE) and down to the left (SW)
IDC_SIZENS	A double-arrow, pointing up and down
IDC_SIZENWSE	A tilted double-arrow, pointing diagonally up to the left (NW) and down to the right (SE)
IDC_SIZEWE	A double-arrow pointing left and right
IDC_UPARROW	Similar to the default cursor, an arrow pointing directly up
IDC_WAIT	An hourglass, or wait, symbol

WARNING

While both the IDC_SIZE and IDC_SIZEALL cursors are documented as producing the identical four-pointed arrow, the IDC_SIZE cursor is now obsolete. For reliability, use IDC_SIZEALL instead.

Regardless of what the default cursor is, applications are free to change cursors at any time, as appropriate to specific tasks or to window areas. They also may define their own custom cursors. The predefined cursor shapes (see Table S3.4) are illustrated in the *Mouse2* demo.

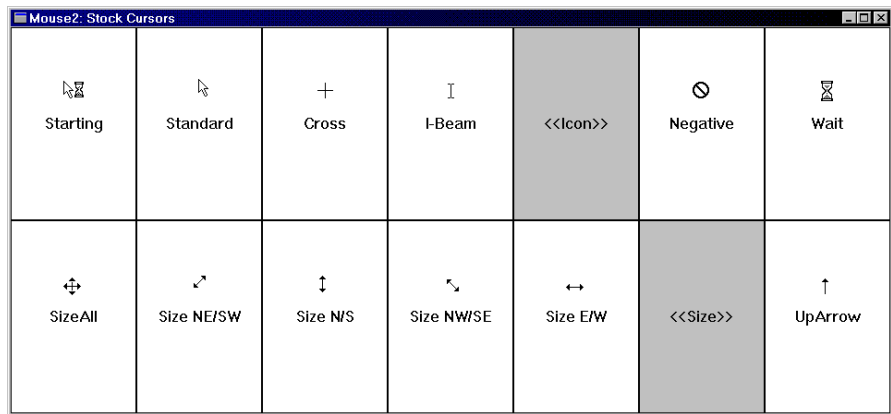
The *Mouse2* Demo: Mouse Cursor Shapes



As shown in Figure S3.1, the *Mouse2* demo subdivides the application client window into child windows, each appearing as a simple outline with a white background. These child windows are secondary to the principal purposes of the example, which are to demonstrate the predefined mouse cursor shapes supplied by Windows 98 and show how the mouse cursor is changed by the application. However, you must create and manage these child windows for the example, as described here. (See Supplement 4 for more details about child windows and control elements.)

FIGURE S3.1:

The *Mouse2* demo shows multiple cursor shapes



NOTE

The *Mouse2* demo is included on the CD that accompanies this book, in the Supplement 3 folder.

Creating the Child Windows

To create the child windows for *Mouse2*, the first step occurs in the `WinMain` procedure. After registering the window class in `WinMain`, the `InitApplication` subprocedure is called from the `Template.I` include file. This subprocedure is then used to make another local set of `wc` assignments to register the child classes.

```
if( ! hPrevInstance )
{
    if( ! InitApplication( hInstance ) )
        return( FALSE );
    /*** also register child window class ***/
    wc.style          = CS_HREDRAW | CS_VREDRAW;
    wc.cbClsExtra     = 0;
    wc.cbWndExtra     = 0;
    wc.hInstance      = hInstance;
    wc.hbrBackground  = GetStockObject( LTGRAY_BRUSH );
    wc.lpszMenuName   = NULL;
    wc.lpfnWndProc    = ChildWndProc;
    wc.hIcon          = NULL;
    wc.hCursor        = NULL;          // essential!!! //
    wc.lpszClassName  = szChildClass;
    RegisterClass( &wc );
}
```

This code includes the three critical `wc` assignments:

- The pointer to the `ChildWndProc`
- The assignment of the `hCursor` field as `NULL`
- The class name, which is a string defined at the beginning of the source code, following the fashion used for the main window class

After you have registered the child class, the main window is created, as usual, by calling the `InitInstance` subprocedure and then entering the message loop. Remember, however, that at this point, only the main window instance has been created; the child window class has been registered, but no instances of this class yet exist.

The actual child windows are created in the `WndProc` procedure, where an array of handles are defined as:

```
static HWND hwndChild[7][2];
```

These child window handles are defined as a static array of handles because the values assigned must remain, even when the application exits from this local procedure. Any nonstatic data may be overwritten between messages. This is fine for temporary variables, but the child window handles, once assigned by the `CreateWindow` function call, must be preserved.

The `WM_CREATE` message is only issued once—when the application is first created. Therefore, 14 child window instances are created at this time, each returning a handle, which is stored in the `hwndChild` array.

```
case WM_CREATE:
    for( x=0; x<7; x++ )
        for( y=0; y<2; y++ )
            hwndChild[x][y] =
                CreateWindow( szChildClass,
                    NULL, WS_CHILDWINDOW | WS_VISIBLE,
                    0, 0, 0, 0, hwnd,
                    (HMENU) ( x | ( y << 8 ) ),
                    (HANDLE) GetWindowLong( hwnd, GWL_HINSTANCE ),
                    (LPVOID) NULL );
```

Here, there are three principal differences between the operation for the child window and the corresponding operation for the parent (application) window:

- Each child window is created using the `hwnd` handle identifying the parent window. For the parent, the corresponding argument was `NULL`.
- Each child window requires an ID value that must be unique within this application. In this case, the ID is generated as a word value (`x|y<<8`) and is typecast using the `HMENU` data type. Although `HMENU` is not the data type you might expect for a window ID, the `CreateWindow` function is also used to create menus, and this is the type definition expected in the function declaration. The equivalent `HWND` type will also work, but it will result in a warning message from the Microsoft C++ compiler. The corresponding ID values will be needed in `ChildWndProc` to identify specific child windows.
- The child window instance parameter is supplied by calling the `GetWindowLong` function with the `GWL_HINSTANCE` argument. When the application's client window was initialized, Windows had supplied this argument as the `hInstance` parameter passed to the `WinMain` procedure. But now, 12 separate and unique instance handles are needed, and they must be supplied by Windows, indirectly or directly.

NOTE

In Windows 3.1, the child window instance parameter is supplied by calling the `GetWindowWord` function with the `GW_W_HINSTANCE` argument.

There are other differences as well, such as the absence of window titles (identified as NULL) and the differences in the style flag, but these will vary depending on the style and type of child windows desired.

Also, these windows have all been created with both size and position set at zero. While the same was done for the main client window, the main window is sized automatically. The child windows, however, will not be defined as a style that will receive WM_SIZE messages. Therefore, to size and space these dozen children, a provision is required to accomplish this anytime the main client window's size changes.

Ergo, in response to the WM_SIZE message, the WndProc procedure begins by using the current window size in the high and low words in lParam:

```
case WM_SIZE:
    cxWin = LOWORD( lParam ) / 5;
    cyWin = HIWORD( lParam ) / 2;
    for( x=0; x<7; x++ )
        for( y=0; y<2; y++ )
            MoveWindow( hwndChild[x][y], x * cxWin, y * cyWin,
                        cxWin, cyWin, TRUE );
    break;
```

Once cxWin and cyWin hold the appropriate size and spacing for each child window, a double loop calls the MoveWindow function using the handles in the hwndChild array and sizes and positions each.

Operating the Cursor in the Child Windows

As the mouse moves from one child window to another in the *Mouse2* demo, the mouse cursor image changes for each window. To accomplish this, however, a bit of subterfuge is required. Initially, when the child window style was registered, the cursor was declared NULL, affecting all instances of the child class.

Now, as the mouse moves, each WM_MOUSEMOVE message is directed to whichever child window the mouse happens to occupy, and the response sets the mouse cursor shape appropriate to that child window. However, there is only one ChildWndProc to handle responses for all of the child windows. To assign the appropriate cursor, the WM_MOUSEMOVE response requires one additional piece of information—which child window is currently being handled.

The `WM_MOUSEMOVE` response begins with an inquiry to retrieve the child window ID, as:

```
case WM_MOUSEMOVE:
    switch( GetWindowLong( hwnd, GWL_ID ) )
    {
```

Once the child window ID is known, the `switch/case` statement can branch to the appropriate response, as:

```
case 0x0000:
    SetCursor( LoadCursor( NULL, IDC_APPSTARTING ) );
    break;
...
case 0x0106:
    SetCursor( LoadCursor( NULL, IDC_WAIT ) );
    break;
```

Remember, the child window IDs were assigned using the formula $(x|(y<8))$, but here it's simpler to just assign the appropriate constants, especially because formulas are not permitted as case statement IDs. Alternatively, a series of mnemonics could have been declared, but this is simple enough for demo purposes.

In the *Mouse2* demo, anytime you press the left (or primary) mouse button, the cursor will shift to the default arrow cursor. Likewise, anytime you press the right (or secondary) mouse button, the mouse cursor will be hidden until the button is released.

Also, the changing cursors are assigned (loaded) only in response to a mouse movement. Once the selected cursor has been replaced by the default cursor, the selected cursor is reloaded only when another mouse-movement message is received. To avoid this type of interruption, the mouse-button messages also need to be handled to ensure that the desired mouse cursor is shown.

Hiding the mouse cursor might be the simplest task of all. It requires only the `ShowCursor` function and an argument specifying whether the cursor is to be hidden or visible.

In Supplement 2, the `ShowCaret` and `HideCaret` functions were used in a similar fashion to hide and reveal the text caret. In that discussion, I mentioned that you could encounter a problem with multiple show or hide calls because they require equal occurrences of their counterparts before any action occurs.

The same holds true for the `ShowCursor` function, except that instead of a corresponding `HideCursor` function, `ShowCursor` accepts a `TRUE` or `FALSE` argument. However, if the `ShowCursor(FALSE)` function is called and the mouse is then moved to another child window before a `ShowCursor(TRUE)` call is made, a different problem may occur: The mouse cursor may not be visible again—at least not until you leave the application window entirely.

To prevent either problem from occurring, a Boolean variable, `Visible`, has been declared. This variable holds the current state of the mouse cursor, and `ShowCursor(FALSE)` is only permitted if `Visible` is `TRUE` and vice versa. A similar solution, if needed, can be applied to the `ShowCaret/HideCaret` functions.

There's also a second solution. The `ShowCursor` function returns the new display count resulting from the operation, and the cursor is hidden any time the count is negative and shown whenever the count is zero or higher; therefore, the following code lines will increment or decrement the show count until the appropriate value is reached:

```
while( ShowCursor(FALSE) >= 0 );      // hides cursor
while( ShowCursor(TRUE)  <  0 );      // reveals cursor
```

NOTE

As an alternative to using predefined cursors, you can include custom cursors in your application. See Supplement 6 for details.

The *Mouse3* Demo: Hit Testing



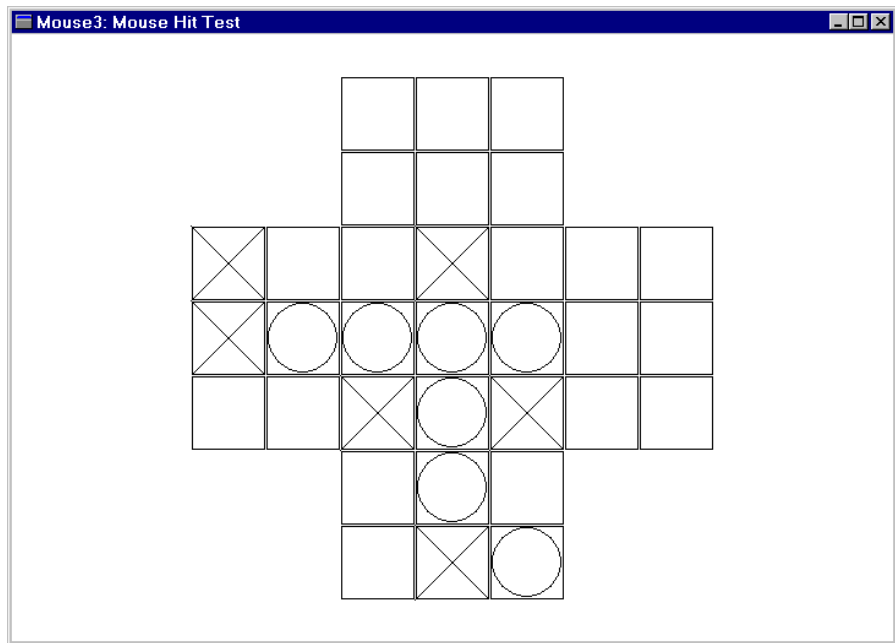
The *Mouse3* demo demonstrates how mouse-click events are registered. It uses a simple cruciform game field, which responds to both the left and right mouse buttons by displaying, respectively, an X or O. Figure S3.2 illustrates the game field.

NOTE

The “game” using this board was originally devised by a young lady of ten. She came up with a rather complex set of rules that, you may be properly relieved to know, are not implemented here.

FIGURE S3.2:

A cruciform playing board
for mouse-hit testing



The actual grid is a 7×7 array, with four elements from each corner flagged as invalid and, therefore, not included in the paint operations. Each remaining grid element has a corresponding integer flag, which is set to 0 initially.

To demonstrate mouse-hit testing, the location of each `WM_LBUTTONDOWN` and `WM_RBUTTONDOWN` message is tested against the grid coordinates and then shifted according to the original state and the button clicked before invalidating the specific grid. Conversely, any mouse clicks that fall outside the grid area or on an array element flagged as invalid produce a warning beep.

As for playing the game, feel free to invent your own rules.

NOTE

The *Mouse3* demo is included on the CD that accompanies this book, in the Supplement 3 folder.

In addition to handling mouse-event messages, the three programs discussed in this chapter demonstrate all the principal mouse operations within the client-window area: tracking the mouse, changing the mouse cursor, and testing mouse hits. Because mouse operations are integral to any Windows application, they will continue to be discussed in later chapter. But for now, we'll leave that topic and turn to some other basic elements of Windows applications, which we touched on briefly in this chapter: child windows and control elements.

S U P P L E M E N T

F O U R

S4

Child Windows and Control Elements

- Types of window controls
- Control button styles
- Control button grouping
- Button event messages

You've already seen examples using child windows. In Supplement 3, two of the programs we discussed included child windows: one to demonstrate different system cursors, and another to respond to mouse-button events by displaying either crosses or circles in child windows.

Generically, child window controls are windows that process mouse and keyboard messages. They handle their own responses appropriately and notify the parent window in some suitable fashion as necessary when a control changes the child window's state (for example, a different radio button has been chosen or a scrollbar has been adjusted).

As you will be reassured to know, you can normally accomplish these tasks without the need for the elaborate provisions shown in the `ChildWndProc` procedures in the demo programs in Supplement 3. Instead, the usual child windows are predefined window classes, several of which will be demonstrated in this chapter and all of which will be demonstrated in future chapters.

Before moving on to Part 2, where you'll learn about an easier way of using window control elements, you need to understand how child windows are handled using direct, rather than indirect, interactions, which is the topic of this chapter.

The Programmer and Child Window Controls

The real use of child windows is in the form of child window controls, although they are not usually referred to as such. Instead, child window controls are commonly referred to by their functions: as buttons, checkboxes, radio buttons, edit boxes, list boxes, scrollbars, and so on.

NOTE

We've talked about scrollbars in earlier chapters, but only in one of their many possible forms. As you will see later, scrollbars can be used for a variety of purposes other than scrolling windows.

If you are familiar with Windows, you've already encountered a wide variety of Windows control elements and know how convenient they are for the user.

Equally important is just how convenient these control elements are for the programmer. The program (and, therefore, the programmer) does not need to be concerned with the mouse driver and mouse-button logic or with any of the other myriad details involved in using these controls, such as making control buttons change state or adjusting scrollbars when they are dragged. Instead, as the programmer, you are free to use them “as is,” as the end user does.

As a programmer, the extent of your involvement is simple. You just define the control elements needed and the appropriate responses—and then wait. When a control is activated, a `WM_COMMAND` message is returned, together with additional information identifying which control and, if appropriate, specifics about the state of the control.

For example, in the *Mouse2* demo (discussed in Supplement 3), a child window class is defined and registered (in this case, in the `WinMain` procedure). Then, in the `WndProc` procedure, this child window class is initiated, as individual instances, using the `CreateWindow` function before positioning and sizing each using the `MoveWindow` function.

The *Button1* and *Button2* demos described in this chapter use a similar process, calling the `CreateWindow` function to create instances of predefined classes before positioning and sizing each using the `MoveWindow` function. For the predefined window classes, such as buttons and scrollbars, the process becomes much simpler. For these window controls, the classes and their responses are already defined, programmed, and compiled, and they are available simply as library functions.

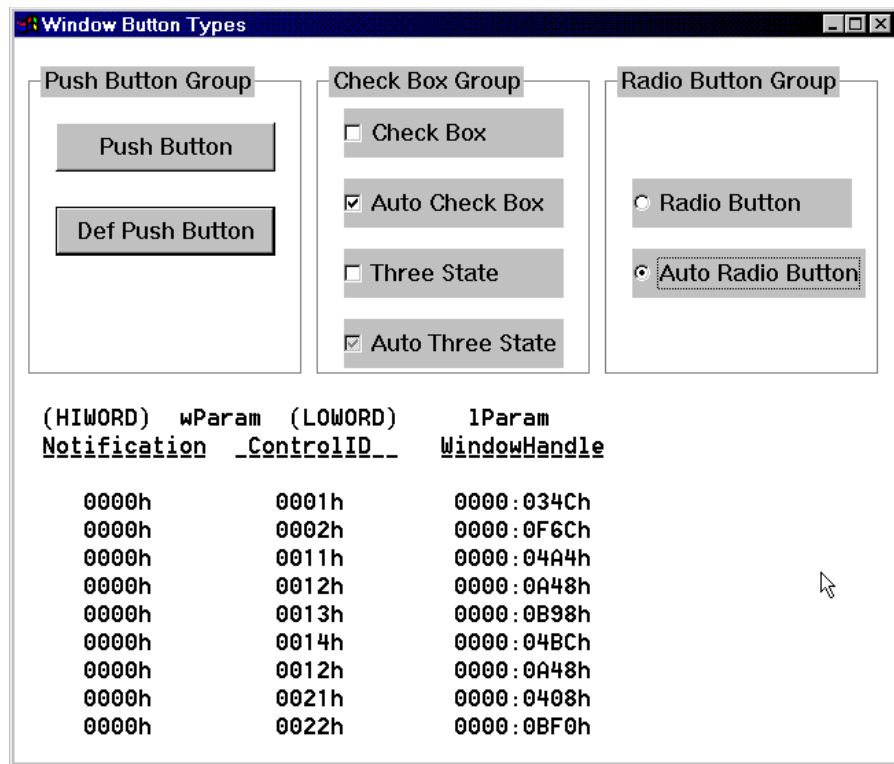
The third demo described in this chapter, *Button3*, duplicates the codes returned by the child window controls, but uses MFC to create a dialog box-based application and introduces a wider variety of button styles.

Control Button Types

Windows provides three principal types of control buttons: pushbuttons, checkboxes, and radio buttons. Figure S4.1 shows several examples of each of these three types.

FIGURE S4.1:

Standard button
control styles

**NOTE**

Another child window type also appears in Figure S4.1. A *group box* contains each of the sets of examples. However, group boxes do not respond to mouse events or issue WM_COMMAND messages. They are used only to visually group other control elements (with or without group labels).

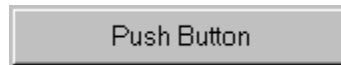
Because these child window buttons are “drawn” on a regular white window background, rather than on the half-tone gray background common to dialog boxes, here the controls appear unfinished. Because the purpose is to demonstrate messages from child windows, the incomplete appearance shouldn’t detract from the buttons’ functions. (For a more polished version, see the discussion of the *Button3* demo, later in the chapter.)

Pushbutton Styles

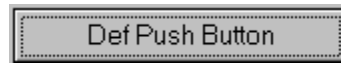
Generic pushbutton controls are rectangular boxes that have a centered text label and an outline simulating a raised button (3-D shading). When activated, such as by a mouse click, the outline changes to simulate a button that has been physically depressed. This type of control button is commonly used to initiate immediate actions, without retaining or displaying any continuing status information.

For control buttons, the entire active area is enclosed by the button image, although the control button may be any size desired. Two types of pushbutton controls are predefined:

BS_PUSHBUTTON A control button displaying an optional text label. The pushbutton posts a message to the parent window when activated, while briefly changing state to simulate being physically pressed.



BS_DEFPUSHBUTTON A control button similar to the BS_PUSHBUTTON control, but with a heavy border. This button represents the default response and normally accepts the Enter key as equivalent to being pressed. (The heavy border represents the control that currently has the focus.)



NOTE

Another Windows button type, **BS_PUSHBOX**, was defined previously in Windows 3.x. The **BS_PUSHBOX** style appeared initially as only a text label without a button outline. When selected, the button outline appeared (as per **BS_PUSHBUTTON**) and the label was highlighted; it remained so until another pushbutton or control was selected and the input focus was lost.

Checkbox Styles

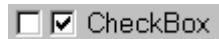
Checkboxes are small squares with text labels appearing to the right by default (but you can change them to the left with **BS_LEFTTEXT**, as explained in the “Special

Controls and Modifiers” section). The checked state is shown by a checkmark (or by an X in MFC) in the box.

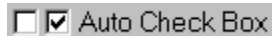
Although the box element of the checkbox is fixed in size, the active area, including the optional label, can be any size desired (within practical limits). To select the button, the user can click the mouse anywhere within the checkbox window, not just on the checkbox proper, whether or not the region is visibly delineated.

Four checkbox styles are defined:

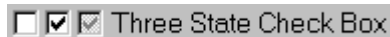
BS_CHECKBOX A checkbox that displays a bold border when the button is checked. The button state must be set by the owner (application) and is not displayed automatically.



BS_AUTOCHECKBOX The same as BS_CHECKBOX, except that the button state is automatically toggled when selected or deselected.



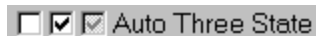
BS_3STATE The same as BS_CHECKBOX, except that three states can be selected: clear, checked, or grayed. The button state must be set by the owner (application) and is not displayed automatically.



NOTE

The grayed state is typically used to show that a checkbox has been disabled.

BS_AUTO3STATE The same as BS_3STATE, except that the checkbox automatically steps through the three states in clear, checked, gray-checked order (unless otherwise explicitly set by the application). In the third state, the checked state appears as a dark gray checkmark (or, using MFC, an X) against a lighter gray square.



The state (setting) of any of these checkbox controls can be queried or set directly using the mouse.

Radio Button Styles

Radio buttons are small, circular buttons with optional text labels that appear to the immediate right by default (like checkbox labels, radio button labels can also be displayed on the left using `BS_LEFTTEXT`, as explained in the next section).

By custom and intention, radio button controls are used as groups representing mutually exclusive choices. Only one button in a group can be selected at any time. A set radio button is shown with a solid center. Selecting a radio button a second time (once it has been chosen) does not change the button status. Normally, a default or initial choice is selected when the group is initially displayed.

TIP

MFC also offers the option of creating radio buttons without mutual exclusivity—the equivalent of placing each radio button in its own group.

Two styles of radio buttons include:

BS_RADIOBUTTON A radio button that displays a bold outline when clicked, but does not display a set condition or clear set condition until explicitly directed by the owner (application).



BS_AUTORADIOBUTTON The same as `BS_RADIOBUTTON`, except when a button in a group is selected, a `BM_CLICKED` message notifies the application, the selected button is set automatically, and all other auto radio buttons in the group are cleared automatically. Any non-auto radio buttons within the group will not be affected without explicit instructions from the application.



The state (setting) of either type of radio button control can be queried directly by the application or can be explicitly set as required.


Special Controls and Modifiers

Three additional BS_XXXX button types are defined for special purposes:

BS_OWNERDRAW This designates an owner-drawn button, but it does not provide any type of image or response. Instead, the parent window is notified when the button is clicked and is expected to supply provisions to paint, invert, and/or disable the button using application-supplied bitmap images.

BS_GROUPBOX This designates a rectangle used to visually group other buttons with or without an optional label that will be displayed in the rectangle's upper-left corner. The position and size for the group box must be specified appropriately to enclose the controls or area desired. The group box does not respond to mouse events or return any WM_COMMAND messages. (Figure S4.1, shown earlier, includes three group boxes.)

BS_LEFTTEXT This designates a flag used in combination with the BS_CHECKBOX, BS_RADIOBUTTON, or BS_3STATE style to shift the label to the left of the checkbox or button. This flag is not valid with pushbuttons.

Radio #4 

The Button1 and Button2 Demos: Button Operations



The button operations demonstrated in the *Button1* and *Button2* demos are less flexible than the usual standards typified by dialog-box button operations, because the only provisions included for these child window operations are to respond to the mouse. No provisions have been made to permit control selections using the Tab or cursor keys, nor do any of these, including the default pushbutton, respond to the Enter key.

Likewise, except for the BS_AUTOCHECKBOX, BS_AUTO3STATE, and BS_AUTORADIOBUTTON styles, none of the controls demonstrated display any change of state beyond the immediate selection “flash” when clicked with the mouse.

NOTE

Although this is not obvious in either demo, the child window controls demonstrated can obtain the input focus when selected with the mouse. However, they do not subsequently release the input focus to the parent window.

When selected, any of the buttons demonstrated send a `WM_COMMAND` message to the parent window. In the *Button1* demo, this message is displayed in a table below the buttons as a breakdown of the `wParam` and `lParam` arguments. (You can see this in Figure S4.1, shown earlier in the chapter.) The table shows (from left to right):

- The notification value (high word in `wParam`)
- The child window (control) ID (low word in `wParam`)
- The child window handle value (`lParam`)

You will learn all about notification values in the section “Button Control Communications: A Two-Way Channel,” later in this chapter.

NOTE

The *Button1* and *Button2* demos are included on the CD that accompanies this book, in the Supplement 4 folder.

Using CreateWindow for Buttons

The individual control buttons are generated using the same `CreateWindow` function that we have used to create an application’s client window and, in Supplement 3, to create a series of child windows. This time, however, the `CreateWindow` function is going to be used in a somewhat unusual fashion—to create sets of buttons grouped together by a `BM_GROUPBOX` child window.

The parameters used in calling `CreateWindow` for this purpose are defined as:

lpszClassName ASCIIZ character string identifying the window class. This class may be a predefined window class or a registered custom class. Note that an error in this field does not cause a compiler error but will not allow an erroneous class object to be displayed.

lpSzWindowText Pointer to an ASCIIZ character string, providing a label for the button or control.

dwStyle Double-word style identifier, which uses the predefined window and control styles.

x Integer specifying the initial x-axis position of the button class (relative to the parent window origin).

y Integer specifying the initial y-axis position of the button class (relative to the parent window origin).

nWidth Integer value specifying the control button width (in device units).

nHeight Integer value specifying the control button height (in device units).

hWndParent Parameter that identifies the parent window (owner) of the window or control being created.

hMenu Unique value identifying the child window (in other circumstances this value may identify a menu belonging to the window; the meaning depends on the style definition). Notice that the values used for this field are always cast as HMENU types, regardless of their intended function. This cast is necessary to prevent a compiler warning but does not affect the actual operation.

hInstance Parameter that identifies the instance creating the window or control.

lpParam Pointer used to address extra parameters or, in the *Button1* demo, to chain a series of window controls. The chain is terminated by passing the final *lpParam* argument as NULL.

The actual code is rather unwieldy, making the chain structure of these grouped controls difficult to follow. Therefore, the following list omits most of the parameter arguments in favor of illustrating the links in the declarations.

```

hwndGroup =
    CreateWindow
    (
        ... , ... , ... , ... , ... ,
        ... , ... , hwnd, ... , ... ,
        CreateWindow
        (
            ... , ... , ... , ... , ... ,
            ... , ... , hwnd, ... , ... ,

```



```

CreateWindow
(
    ... , ... , ... , ... , ... ,
    ... , ... , hwnd, ... , ... ,
    CreateWindow
    (
        ... , ... , ... , ... , ... ,
        ... , ... , hwnd, ... , ... ,
        CreateWindow
        (
            ... , ... , ... , ... , ... ,
            ... , ... , hwnd, ... , ... ,
            NULL
        )
    )
)
)
)
)
)

```

In this diagrammatic example, the `hwndGroup` contains five control window elements that are chained together in the declaration. The topmost element in this group is the declaration for the group box itself (though this is not an iron-clad requirement), while the first control button appears as the eleventh parameter in the group-box declaration, and so forth.

While there are no firm limits—memory and system limits aside—on such recursive declarations, this programming style is awkward. It may appear to produce a result that is not actually accomplished; contrary to what the structure seems to suggest, these control buttons are not grouped by the group box except visually, both on screen and in the program.

Grouping Controls

Grouping is not accomplished by the declaration tree (nor by the screen appearance). Instead, a group is declared by first calling `CreateWindow` to create the group box window and then by using the group-box handle (`hwndGroup`) as the owner of the group members when these are declared.

In the tree shown previously, all the elements in the declaration tree used the same parent window handle—that of the client window—by necessity.

In contrast, the *Button2* demo actually does create groups with member controls that are declared and created in the appropriate fashion, even if this is less elaborate.

Here is the *Button2* code in skeleton format:

```

hwndPB[0] =
    CreateWindow
    ( ... , ... , BS_GROUPBOX, ... ,
      ... , ... , ... , hwnd,
      ... , ... , NULL );
for( i=1; i<4; i++ )
    hwndPB[i] =
        CreateWindow
        ( ... , ... , BS_xxcontrolxx, ... ,
          ... , ... , ... , hwndPB[0],
          ... , ... , NULL );

```

Here, the first step returns a handle to the group box control, which has the application's client window as a parent/owner. This handle is then used, in the second step within the loop, to provide the parent/owner for the individual control instances. This format creates four child window controls. These controls belong to the group box and, therefore, are isolated from other controls—a necessary requirement for auto radio buttons, for example.

However, there's still a fly in the ointment! Why? Because this format, while satisfactory for the present demo purposes, is still not practical for general applications.

While the parent window, `hwndPB[0]`, is a group box and does receive messages from the child window processes, it has no provisions to act on these messages and cannot forward them to any other process for action. Ergo, the only actions and responses in the *Button2* demo are those inherent in the button classes themselves.

The real solution is also quite simple and is used, in part, in the *Mouse2* and *Mouse3* demos (discussed in Supplement 3), when child window processes were first introduced. If you recall, in addition to creating the child windows used in the programs (refer particularly to *Mouse3*), a subprocess was also created with provisions to respond to messages from the child window. Thus, to use button controls in the fashion illustrated, the group box—whether it's visible or invisible—or another child window process serving as a parent to the controls would be created in similar fashion, complete with the appropriate responses to handle messages from the child window controls (the buttons).

Now, having been lead through this maze of complexity, you can proceed to forget about it—at least for the present—because your applications, and the examples discussed in later chapters, will use a quite different process to create

these and other control elements. For the most part, you will create control elements by using dialog boxes and dialog box editors, not directly within the application code. Be assured, this is a much simpler, as well as a much more practical, route.

Before leaving this topic altogether, there are other aspects of child window controls that will be relevant, regardless of how the controls are created.

Controlling Button Communications: A Two-Way Channel

Although auto checkboxes and auto radio buttons change their state automatically when clicked, you need to be able to set initial states for checkboxes and radio buttons—for both the normal and automatic versions. At the same time, you need to be able to accept notifications from these (and other) controls and to query the status of these controls. These tasks are handled by sending messages to controls and recognizing messages generated by controls, as well as by setting initial states for the controls.

The *Button1* demo demonstrates how event messages are received from the button control elements as `WM_COMMAND` messages with amplifying data in the `lParam` and `wParam` arguments. The `lParam` argument, which will generally be ignored, contains the window identifier for the child window control. More immediately important is the `wParam` argument, which, in the low-order word, reports the control ID assigned by the application when the control was created.

The notification code in the high-order word of `wParam` informs the parent process precisely what event occurred. This notification code consists of one of the six values shown in Table S4.1.

TABLE S4.1: User Button Notification Codes

Code Constant	Value
<code>BN_CLICKED</code>	0
<code>BN_PAINT</code>	1
<code>BN_HILITE</code>	2
<code>BN_UNHILITE</code>	3
<code>BN_DISABLE</code>	4
<code>BN_DOUBLECLICKED</code>	5

The *Button1* demo returns only two event types: 0 or 5. The second of these, `BN_DOUBLECLICKED` is only returned by the `BS_RADIOBUTTON` style control (but not by the `BS_AUTORADIOBUTTON` style control). Control notifications 1 through 4 are returned only by custom control buttons (not illustrated in this chapter) to prompt the parent application to take the appropriate action (if any) to update the control's image.

Sending Messages to Controls

The *Button2* demo demonstrates four group boxes containing groups of child button controls. Unlike in the *Button1* demo, these group boxes are actually parents to the controls enclosed. Because of this, setting an auto radio button in one group will not affect settings in another group, but it will reset other buttons in the same group, as you will see.

The *Button2* demo also demonstrates other aspects involving messages sent from the application to the control buttons. Five button-specific messages are defined in `WinUser.H`, each beginning with the prefix `BM_` (for "button message"), as listed in Table S4.2.

TABLE S4.2: Button Control Messages

Code Constant	Value
<code>BM_GETCHECK</code>	<code>00F0h</code>
<code>BM_SETCHECK</code>	<code>00F1h</code>
<code>BM_GETSTATE</code>	<code>00F2h</code>
<code>BM_SETSTATE</code>	<code>00F3h</code>
<code>BM_SETSTYLE</code>	<code>00F4h</code>

NOTE

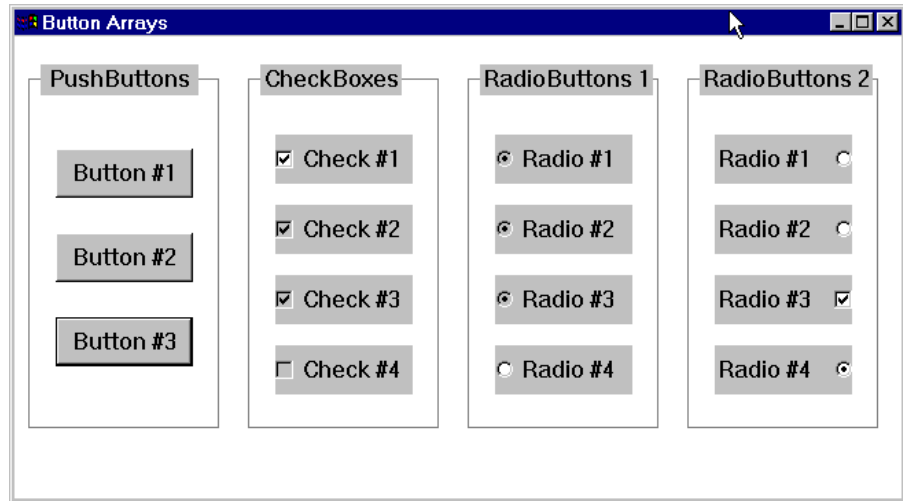
In Windows 3.x, the corresponding message values were defined as `WM_USER+0` through `WM_USER+4`. However, if the defined message constants are used, no conversion will be required to move applications from Windows 3.x to Windows 98.

The `BM_SETCHECK` and `BM_GETCHECK` messages are sent by the parent window to a child window control button to, respectively, retrieve or set the check state of

checkboxes and radio buttons (use with normal, auto, and three-state versions). In *Button2*, the `BM_SETCHECK` message sets three of the checkboxes in the Check-Boxes group, three of the auto radio buttons in the RadioButtons 1 group, and one of the radio buttons in the RadioButtons 2 group. Figure S4.2 shows the *Button2* display.

FIGURE S4.2:

Grouped control buttons



The display in the RadioButtons 1 group, with three auto radio buttons checked, is definitely anomalous and can only occur (short of bad programming) because of the explicit settings made by the application. Clicking any of the buttons in this group will correct the situation shown. The `BM_SETCHECK` message was sent as:

```
SendMessage( hwnd..., BM_SETCHECK, TRUE, 0L );
```

Conversely, to clear the check state of a button, another message would be sent as:

```
SendMessage( hwnd..., BM_SETCHECK, FALSE, 0L );
```

The first parameter is the child window (control button) handle, followed by the message identifier with the third parameter setting or clearing the flag state. The fourth parameter is unused and is passed as zero.

The `BM_SETSTATE` message is used to simulate the button flash that occurs when a button is clicked with the mouse or otherwise activated. In Figure S4.2,

three of the checkboxes (shown with heavy outlines) have received `BM_SETSTATE` messages, which were sent in the same fashion as the `BM_SETCHECK` messages.

The third “set” message provides a means of changing the control button style during execution. *Button2* demonstrates this with two examples.

First, the `BM_SETSTYLE` message is used to change the style of the third (bottom) pushbutton from `BS_PUSHBUTTON` to `BS_DEFPUSHBUTTON`:

```
SendMessage( hwndPB[3], BM_SETSTYLE,
             BStyle | BS_DEFPUSHBUTTON, 1L );
```

Unlike the earlier set messages, the `BM_SETSTYLE` message does use the fourth parameter, passing a nonzero argument to request that the control be redrawn immediately, using the new style settings. A zero argument leaves the control unchanged until some other circumstance causes the control to be redrawn.

Second, one more `BM_SETSTYLE` message is sent to change the style of the third radio button in the *RadioButtons 2* group to a checkbox, but, at the same time, the message does not incorporate the `BS_LEFTTEXT` flag originally used with all controls in this group. As a result, the newly styled control button behaves precisely like any other checkbox but remains a member of the *RadioButtons 2* group.

Querying Control States

The `BM_GETCHECK` and `BM_GETSTATE` messages are sent as information requests directed to specific button controls. They return the check or state status as `TRUE` if the button is checked or depressed (that is, the state flag is set) or `FALSE` if the appropriate flag is clear. As an example, the following instruction retrieves the check status:

```
fStatus = SendMessage( hwnd..., BM_GETCHECK, 0, 0L );
```

In this inquiry, only two parameters are relevant: the handle identifying the control element and the `BM_GETCHECK` message.

On the other hand, if the only requirement is to flip the state of a button or checkbox, you can combine the `BM_GETCHECK` and `BM_SETCHECK` instructions:

```
SendMessage( hwnd..., BM_SETCHECK,
             (WORD)SendMessage( hwnd...,
                                BM_GETCHECK, 0, 0L ), 0L );
```

However, this latter form is generally not required because the `BM_AUTOxxxxx` styles obviate the need for the application to handle the button state directly.

Changing Button (Window) Labels

One item that applications may wish to change on buttons and controls of all types—whether to present a different series of selections or to update other information types—are the button labels. This also applies to window captions and the window text for any window type. The `SetWindowText` function is written as:

```
SetWindowText( hwnd, lpszString );
```

The `hwnd` parameter identifies the window, and the string argument is passed as a long or far pointer to an ASCIIZ string. Button labels are constrained to a single line of text and line wrapping is not allowed.

TIP

You can create a multiline button using a static text field for the button text and omitting text entirely from the button. The disadvantage of this approach, of course, would be that the user would need to click the button itself—clicking the text would not set the button. This is true unless the application included additional code to recognize a hit on the separate label and to issue a button message in response.

At the same time, you can retrieve the current text label from any window type using the `GetWindowText` function:

```
nLen = GetWindowText( hwnd, &Buff, sizeof( Buff ) );
```

Obviously, the buffer must be large enough to hold the string information returned. The third parameter places a limit on the length copied. The value returned directly to `nLen` is the actual length copied.

If the length is unknown, the `GetWindowTextLength` function can be called:

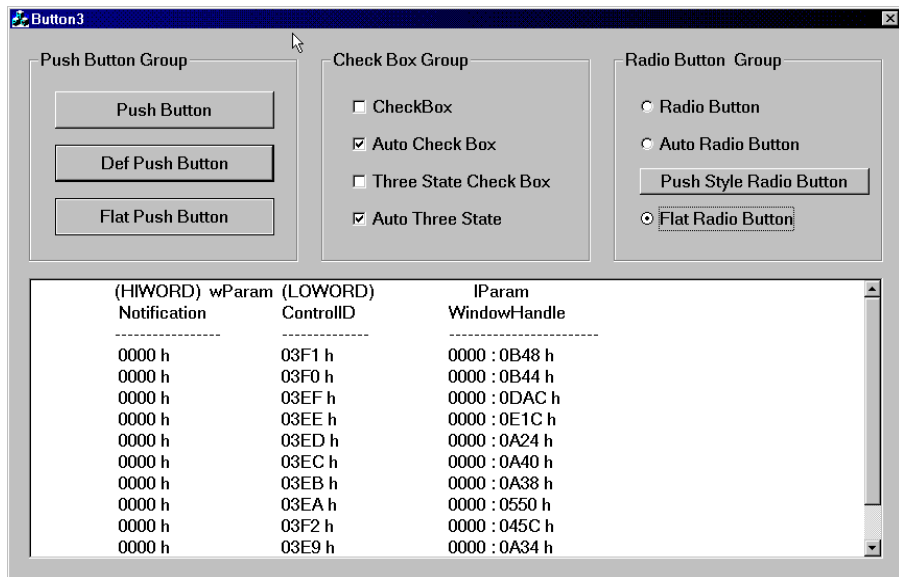
```
nLen = GetWindowTextLength( hwnd );
```

A More Elaborate Version: Button3

Figure S4.3 shows the *Button3* demo, which, like *Button1*, reports the event messages produced when the various buttons are pressed. Unlike the earlier version, however, this variation uses a dialog box–based application window, where the various buttons appear against the appropriate background.

FIGURE S4.3:

A finished button demo



In addition to providing a more complete demo, *Button3* also demonstrates three additional button styles:

- The flat pushbutton
- The push-style radio button
- The flat radio button

These styles are provided by MFC; they are not included in the standard defined button styles.

NOTE

The *Button3* demo is included on the CD that accompanies this book, in the Supplement 4 folder.

Now that you've finished this chapter, you may be tempted to simply forget almost all of it since these tasks can be accomplished in a simpler fashion. However, whether you use these forms directly or not, a clear understanding of how

applications use child window controls is still, if not essential, certainly very valuable. After all, it isn't what you don't know that hurts half as much as what you think you know but actually don't.

In Part 1, you've seen the mechanisms from the inside; in Part 2, you will look at these same mechanisms (and others) from a higher (and easier to implement) level.

S U P P L E M E N T

F I V E

S5

An Introduction to Application Resources

- Types of application resources
- Types of resource files
- Resource manager functions

Resources in Windows applications appear as a variety of elements. These elements range from text-based menus, to dialog box displays combining graphic and text elements, to purely graphic elements that include bitmaps, cursors, and icons. Each individual application may use many or all of these elements. For example, an application might include multiple menus and dialog boxes, dozens of bitmaps, and assortments of cursors representing different operations or modes.

In theory, resource elements can be created as part of an application's conventional executable code. Under DOS, this was essentially what was required. Windows, however, provides a different structural approach for executables. The approach used by Windows permits resources to be appended to an application's executable without being embedded within the operational portion of the program.

This chapter provides an overview of the application resources offered by Windows 98. These topics are then discussed in detail in the following chapters. In Supplement 10, we'll finish up by discussing two versions of an application that uses all the techniques covered in Part 2: the *FileView1* and *FileView2* demos.

Advantages of the Windows Structural Approach

When an application is loaded for execution, the resource elements are not loaded. Instead, resources, such as dialog boxes and menus, are loaded only on demand, as required. When a resource is no longer needed, it is discarded. The advantage of loading resources on demand is simple: Memory is expended only for currently operating elements. Memory is not used for storage of elements that may or may not be needed.

For example, one version of a solitaire program (*So1.EXE*) contains 74 bitmaps (52 for the card faces alone), one menu, five dialog boxes, one accelerator table, one icon, and one custom resource type—a fairly small load as resource elements go.

In contrast, another application might contain only one resource and the application's icon, and then use a series of .DLL files. These .DLL files might contain hundreds of bitmap images, dozens of menus and dialog boxes, multiple accelerator

key sets (for different circumstances), a hundred separate cursors, dozens of additional icons, and a huge string table.

In both cases, if these massive assortments of resources needed to be loaded into memory immediately on execution of the application, your system memory would quickly become overloaded. And only a small portion of these resources might actually be used at any time. Instead, under Windows, because the system loads resources only on demand (when they are needed), memory remains free for other uses.

Another advantage to the Windows approach is that you can edit application resources without needing to recompile the files. While the usual practice is for programmers to edit their own resource files before compiling and linking, they can now edit executable program resources.

NOTE

When executables are opened for resource editing, the original resource names do not appear. Instead, all resources are labeled by their identifier values.

Keep in mind, however, that application resources define only the appearance and organization of the resources, not their functional characteristics. By using a resource editor to edit a dialog box, for example, you can change the arrangement or appearance of that dialog box, but you cannot alter how the application responds to the dialog box controls. This means that you can make only cosmetic changes by editing resource elements. If you need to make functional changes, you must revise the application's source code and then recompile it.

Types of Resources

For your Windows 2000 applications, the following resources are available:

Images Three types of image resources are supported as bitmaps (.BMP), cursors (.CUR), and icons (.ICO). Each of these are bit images, but different rules and organizations are applied to create each one. You can edit image resources with a bitmap editor. The three bit-image resource types are discussed in Supplement 6.

On the Use of Resources

Because my tech reviewer has raised a few important questions concerning limitations on resources (there really aren't any), I'm repeating the questions here with approximate answers.

Q: How much free memory can be used for resources?

A: All of it. In theory, the only limitation—under any 32-bit version of Windows—is the 4GB limit on addressable memory. Remember, that the operating system uses roughly 12MB of RAM, *but* the swap file acts as an extension to system RAM. Therefore, on a 16MB system, somewhere in the neighborhood of 20MB–30MB of RAM are available for resources. How much you use is entirely up to your application design. (And, yes, you could use a 2GB or 3GB hard drive as a dedicated swap file—why not, they're cheap now and relatively fast if you really need that much space.)

Q: List everything that is considered a *resource*.

A: List everything that isn't provided directly by your C/C++ source code. A resource—in addition to bitmaps, icons, toolbars, dialog templates, hotkey accelerators, string tables, and custom cursors—may include data objects, such as a database template, custom controls, default registry data, sound files (but these are usually external), or anything else you desire to include as a custom resource.

Q: How time-consuming is it to load and unload resources?

A: In actual fact, it isn't. Resources are part of the application's .EXE file (or .DLL library) and are loaded at the same time the executable runs. If there is insufficient free memory, some part of the executable or DLL is transferred to the swap file. This task is handled by Windows on a demand basis such that currently unneeded elements are off-loaded to a disk image of RAM and are recalled (moved back into active memory) when needed. In effect, there is no real way to say what the time constraints are except to observe that fast hard drives are more responsive than slow ones.

Q: Besides memory, what are the limitations on how many resources can be loaded at one time?

A: For all practical purposes, memory aside, there are none.

Toolbars These resources are specialized bitmap images consisting of one or more individual button images. By default, each button is 16x15 pixels, but you can size buttons as desired. (While toolbars are bitmap images, most resource editors provide a toolbar editor for greater convenience.) Toolbar resources are discussed in Supplement 6.

Dialog boxes These resources are generally message or input windows, but they may also be child windows used to organize a display. A dialog box editor provides an interactive means of constructing dialog boxes and showing the elements (list boxes, buttons, edit boxes, scrollbars, and so on) exactly as these will appear on screen. Dialog boxes are discussed in Supplement 7.

Menus These resources provide lists of program options. The options may immediately execute commands, display submenus, or display dialog boxes for other operations. A menu editor allows you to define and test main and pull-down menus. You can also create menu resources using any plain-text editor (Windows Notepad, for example). Menus are discussed in Supplement 8.

Accelerators These are keyboard resources. An accelerator resource is a key or key combination provided as an alternative to an individual menu item to invoke a command. One common example is pressing the Shift+Ins or Ctrl+V combinations as an alternative to pulling down the Edit menu to select the Paste option. You can define these hotkey shortcuts for menus with an accelerator editor or a plain-text editor. Accelerator resources are discussed in Supplement 9.

Strings These resources are text strings that are displayed by an application in its menus or dialog boxes, for error messages or other information. By defining text strings as resources, rather than embedding them in the source code, you can conserve memory. Also, keeping all message strings in a single location makes it easier to standardize message formats and to maintain consistency. Another advantage of this approach is that it allows you to edit strings for language changes without recompiling. You can create string tables using any plain-text editor or a string table editor. String resources are discussed in Supplement 9.

Version This resource contains information about the application, such as its version number, its intended operating system, and its original filename. It is intended for use with the File Installation library functions. Version resources are discussed in Supplement 9.

While application resources are, nominally, contained in the .RES resource script file during development, binary resource objects—such as bitmaps, cursors, and icons among others—are stored separately as individual files. Also, several custom file types can be used to store individual resource objects separate from the resource script.

Files and File Types

Most resource editors can create, import, export, or edit most resource files used by Windows, including executable files containing resources. A full list of standard resource types appears in Table S5.1.

TABLE S5.1: Standard (Predefined) Resource File Types

File Extension	Type	Description
.EXE	Executable	Executable program code containing application resources and compiled program code
.RES	Resource	Compiled (binary) resource file
.DLL	Executable	Executable (dynamic link library) module, which may contain either executable code, application resources, or both
.H	Source code	Header file containing symbolic names for defined resources
.ICO	Resource	Individual icon-image resource file
.CUR	Resource	Individual cursor-image resource file
.BMP	Resource	Individual bitmap-image file
.DLG	Resource script	Individual dialog box resource script containing a single dialog box in ASCII text format
.RC	Resource script	ASCII resource script containing one or more resource elements, which may include image resources in hexadecimal format
.DRV	Device driver	Compiled device driver, which may contain resource elements, dialog boxes, and so on
.FON	Font library	File containing one or more fonts belonging to a single type-face (not commonly used as a resource element)

Continued on next page

TABLE S5.1 CONTINUED: Standard (Predefined) Resource File Types

File Extension	Type	Description
.FNT	Font typeface	File containing a single typeface font
.DAT	Resource	Raw data resource, which is used for custom resource types (can be copied, renamed, or deleted but cannot be edited, browsed, or created using a resource editor)

Linking Resources

Normally, the resources are compiled directly to an .RES (binary) file, permitting the Linker to link the compiled resources with the compiled .EXE executable. However, when editing an existing .EXE or .DLL source, no .RES file is created. Instead, the resources are written directly to the runtime program.

Using the Microsoft command-line compiler, NMake scripts (.MAK) contain instructions to compile .RC resource scripts before linking the resulting .RES compiled resources.

Note that both resource editors (and most other systems) create an .RC resource script file—a text file—that contains all of the nonimage resources. Image resources, such as bitmaps, icons, cursors, and toolbars, are normally stored as separate image files referenced by the .RC resource script.

Dynamic Link Libraries

A *dynamic link library* (DLL, sometimes pronounced “dill”) is an executable module that may contain both application executables (compiled source code) and application resources. A DLL is similar in construction to a runtime library, except that it is not linked to the application during the compile process. Instead, DLLs are dynamically linked during execution when library resources—either executable routines or resources—are required.

DLLs have two important strengths:

- A single DLL can be accessed by more than one application without being duplicated within each application.

- Routines in DLLs can be revised and recompiled without modifying the programs using the called routines (assuming, of course, that the call-and-response format remains unchanged).

NOTE

Just as .EXE resources can be modified without recompilation, .DLL resources may also be edited, extracted, or updated without recompilation. Unfortunately, the current Visual C/C++ compiler does not support opening resources from .EXE or .DLL sources, although other compilers have managed this task without difficulty.

Header Files

All application resources must be identified by numeric values. But, for humans, numeric identifiers are awkward and difficult to remember. Therefore, just as Windows 2000 supplies mnemonic constants (see the Windows.H and included header files), programmers can also define .H header definition files to provide mnemonics for application resources (or use predefined mnemonics).

When creating resources using the Microsoft C++ integrated development compiler, the resource identifiers are created automatically and are found in the Resource.H header. The bulk of the resources appear as scripts in the .RC resource file. All image resources—bitmaps, cursors, and icons—appear as separate .BMP, .CUR, and .ICO files, which are referenced within the .RC script.

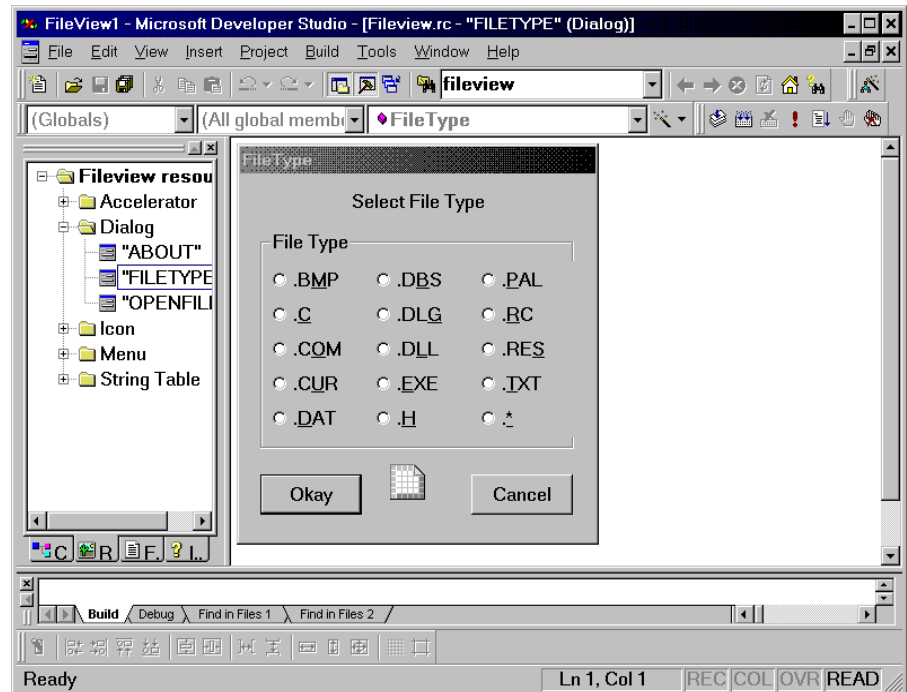
Using a Resource Editor

Although some resource types, such as dialog boxes, can be designed (however laboriously) by editing script files, image-based resource types are difficult to create without using some type of resource editor. In this part, we'll use the resource editor integrated into the Microsoft Developer Studio for our examples. If you prefer, you can select from a variety of other resource editors, all of which produce compatible application resources—both image-based and other types.

Figure S5.1 shows the Microsoft Developer Studio's main screen and primary menu together with the FileType dialog box open in the editor (the resource list appears to the left in the Developer Studio window).

FIGURE S5.1:

The Microsoft Developer Studio main screen



Like most Windows applications, the Developer Studio is designed principally for mouse operation. You can activate features by clicking menu options, buttons, or other controls. However, the Developer Studio also provides hotkey options that allow you to select items by pressing the key corresponding to the under-scored letters. Thus, from the menu, Alt+F selects File, N selects New Project, and a pop-up dialog box or a submenu appears, offering a selection of preferences for creating a new file, a new project, and so on. Similar hotkeys are available in most dialog boxes and menus.

As with many other Windows applications, you can also use the Tab key to cycle through the buttons and/or fields. To select the highlighted option, press the Enter key or spacebar.

Opening Project and Resource Files

Because you can create and store resource elements separately from a project (but most commonly, they will be within a project), the Developer Studio makes provisions for opening both project and individual resource files. The Developer Studio's File menu has two "open" provisions: Open and Open Workspace:

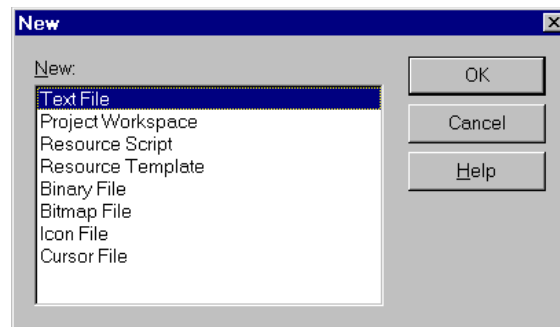


You can use the Open option to open any type of file, including a project file. But the Open Workspace option provides a shortcut specifically for opening projects. Likewise, the Close Workspace option closes an open workspace and all associated files. When you reopen the workspace, all the files that were previously open are reopened.

On the other hand, if you want to work on an individual resource file or create a new resource file, without opening a project, simply click New to open the New dialog box, shown in Figure S5.2, and select the file type.

FIGURE S5.2:

Creating a new file or resource

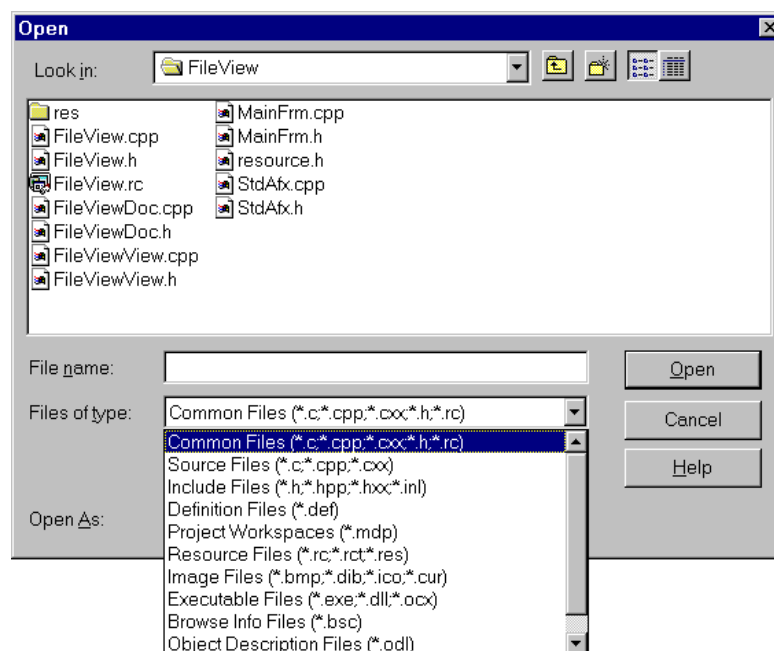


Notice here that there are individual options for bitmap, cursor, and icon files but not for dialog boxes, menus, accelerator keys, and other resources. Instead, all resource types except image resources are covered by the Resource Script option.

If you wish to open an individual resource file, select the Open option to see the dialog box shown in Figure S5.3. Then you can select the type of resource (or other file) from the Files of Type pull-down list.

FIGURE S5.3:

Opening a file outside a project

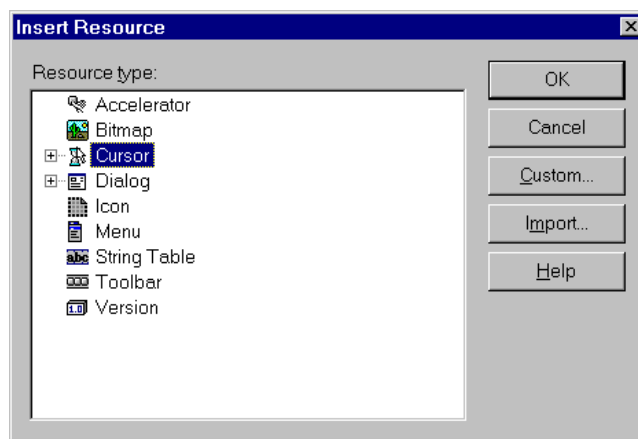


Adding and Editing Resource Elements

Normally, you will be working with a project, and you will want to create your resources as a part of the project rather than as individual resource files. For this purpose, instead of opening a new file, select Insert from the main menu, select Resource, and then choose the type of resource to create from the Insert Resource dialog box (by double-clicking the resource, or highlighting it and clicking OK), as shown in Figure S5.4.

FIGURE S5.4:

Adding a resource to
a project



After you select the resource type, a new resource element is added to the resource list (on the left side of your screen) using a default type name, such as IDC_CURSOR1. At the same time, the appropriate editor is called to create the new resource element.

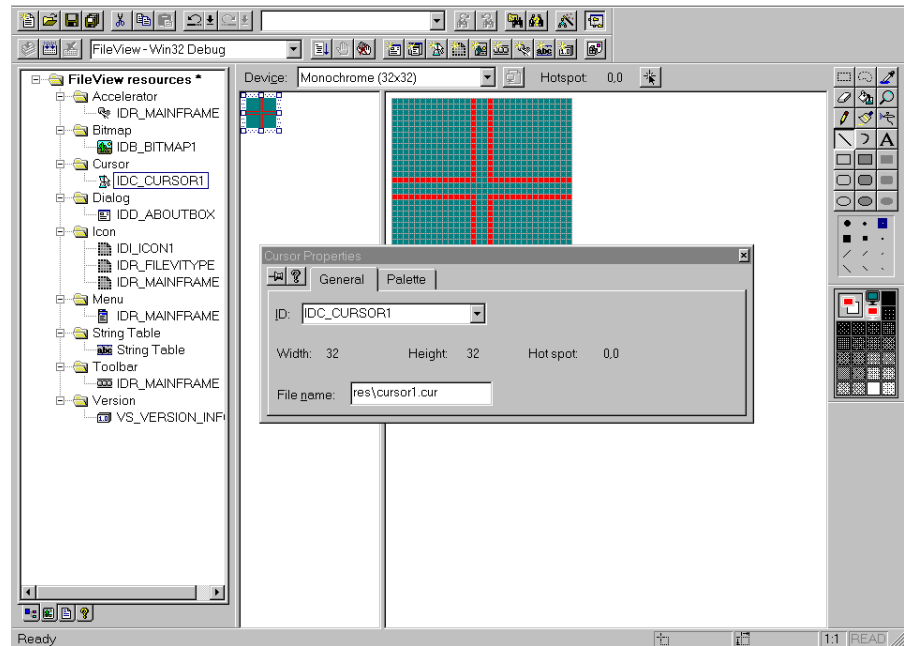
To create a custom resource element, select the Custom button from the Insert Resource dialog box, and then enter a name for the custom resource type. You must create custom resources independently, by whatever means are appropriate. The options here only permit you to include a custom resource and custom resource type within a project. You must handle all other provisions yourself, and they must conform to a response code within the application or an associated DLL.

You can change default element type names at any time by right-clicking the appropriate element in the resource list (on the left side of your screen) to call the pop-up menu. From the menu, select Properties to display the Properties dialog

box, shown in Figure S5.5. Then simply enter a new resource name. A corresponding entry will be made in the `Resource.H` header automatically. This provides a convenient way to replace the default labels supplied when resource elements are generated with new mnemonic resource names.

FIGURE S5.5:

Entering a new resource name in the Properties dialog box



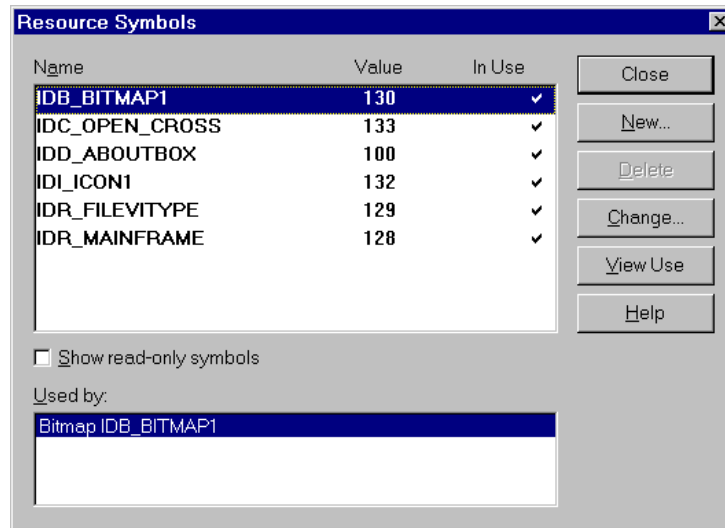
You can edit an existing resource element simply by double-clicking it (or highlighting it and pressing Enter) in the project's resource list. Doing this brings up the appropriate editor and loads the resource element.

Viewing and Changing Resource Identifiers

To view resource identifiers, select Resource Symbols from the View menu. The Resource Symbols dialog box appears, as shown in Figure S5.6. Here the resource identifiers are listed in alphabetical (not numerical) order.

FIGURE S5.6:

Viewing resource identifiers



A checkmark in the In Use column on the right side of the dialog box indicates that an identifier is in use. The Used By box, at the bottom of the dialog box, shows where the highlighted identifier is used. When an identifier is used by more than one resource, multiple items appear in the Used By list. To view the use of an identifier, click the View Use button.

WARNING

Obsolete or unused identifiers may remain in the Resource.H header but will not be checked. You can delete or change them; however, you should be aware that unchecked identifiers may be mnemonic constants created for special messages. The lack of a checkmark simply means that the constant is not used by any resource element.

To change an identifier, click the Change button. If the selected identifier is in use, you will not be able to change that identifier through this dialog box. You can only change an identifier that is in use by changing the properties for the resource element. If the selected identifier is in use, the Delete button will be disabled. You can delete only identifiers that are not in use.

To edit the properties for most resources, right-click the resource item in the tree display, and then select Properties from the pop-up menu. You can then change

the resource properties and identifier in the Properties dialog box. For string table entries, you can change the resource identifiers directly in the string table.

Copying Resource Elements

To duplicate a resource—for example, to use as the basis for creating a different version of the resource—select the resource to duplicate, and then use the Copy and Paste options on the Edit menu. A new resource identifier will be created for the duplicated resource.

The Insert menu also has a Resource Copy option; however, this option creates a copy of a resource element that is used only when a special condition is defined, or it creates a copy in a second language.

Managing Project Resources in the Borland C++ Builder

The Borland C++ Builder has adopted a quite different development approach from previous versions of the compiler (and other compilers), treating projects as containers for one or more applications (presumably associated), while treating applications as constructs largely built from standard components. This approach is similar to Delphi and Visual Basic application design.

The File menu offers options for creating or opening applications, application element files, and projects. Once a project is opened (or created), instead of a resource script, you will find yourself working with “forms,” where each form is associated with a separate source file.

Borland’s C++ Builder also offers a resource editor, but it is difficult to use with conventional C++ or MFC-based C++ application designs. This is because of the development approach taken by the C++ Builder, with its form-centric design and almost total dependence on predefined Borland classes and component-based programming.

As you learned in this chapter, application resources and resource editors can simplify your application development work. The individual resource types and editors are discussed in the following chapters. You’ll also find explanations of how each type is used to create the resources for the *FileView1* and *FileView2* demos discussed in Supplement 10.

S U P P L E M E N T

S I X

S6

Bitmaps, Toolbars, Icons, and Cursors

- The four types of image resources
- Toolbar editing
- Icon design
- Cursor elements

Bitmap and toolbar images, icons, and mouse cursors are image resources, which you can manipulate with an image editor. This chapter describes the differences between these types of image resources and how to work with them.

Types of Image Resources

Although the four types of image resources are bit images, different rules and organizations apply for their creation.

Bitmap Images

The simplest bit-image type is the bitmap image, which is a pixel image using 2, 16, or 256 colors. Individually, bitmaps may be as large as the full screen (or larger) or as small as a few dozen pixels for a checkbox control or radio button; of course, they may also be medium-sized, as in the Solitaire card images. Bitmap images, however, are limited in that they consist only of a foreground image and cannot contain transparent areas.

Bitmap images can be created by any paint program and do not differ in any respect from conventional bitmaps. This means that you can import bitmap images from external sources. Furthermore, bitmap images can use palettes supporting 2, 16, or 256 colors with no limitations (aside from memory, of course).

Within applications, you can use bitmap images for decorative or informative purposes. You can also incorporate them as graphic controls.

Toolbar Images

Toolbars are a specialized bitmap-image format in which a long narrow image is subdivided into individual button images. The bitmap itself is a single continuous image, and the division into individual buttons is strictly an artifact of how the bitmap is presented.

To create a toolbar bitmap, the image editor begins with a blank bitmap the size of a single button. As one button image is created (edited or drawn), the editor adds a new blank to the end of the bitmap image. These blank button images do not appear on the actual bitmap but can be selected to create a new button image.

By default, toolbar buttons are 16×15 pixels. You can specify a new button size in the button properties, but keep in mind that changing the size of any of the buttons on a toolbar affects all of the toolbar buttons. All buttons on a toolbar must be the same size, both vertically and horizontally.

Icon Images

The icon image type is commonly used to represent an application within a Program Manager group or on the Desktop display. You can also include icons as bitmap images within dialog boxes or, with special handling, as menu elements.

Icon images are similar to bitmap images, but they have size limitations supporting only VGA resolution 32×32 (normal) or 64×64 bit image sizes. Like bitmaps, icon palettes can support 2, 16, or 256 colors. Unlike bitmap images, however, icons can include transparent areas or areas that interact with the Desktop or other underlying images by inverting the background pixels.

Cursor Images

The fourth bit-image type is the cursor, which, unlike the bitmap, toolbar, and icon types, always interacts with the background image.

A cursor bit image consists of two 32×32 bit images: a mask that interacts with the cursor background and the cursor pattern itself, which overlays the mask/background combination. Cursors are restricted to a default palette consisting of only four colors: black, white, transparent, and inverted.

Also, unlike other images, a cursor contains a *hotspot*, which is a pixel location within the image that defines the cursor's position. For example, the familiar arrow cursor's hotspot is located at the tip of the arrowhead.

Unfortunately, animated cursors (.ANI) are not supported by present resource editors—even though Windows 98 ships with a set of sample animated cursors—and require specialized facilities to create. Animated cursors are discussed in more detail later in the chapter.

Custom Fonts

At one time, custom fonts represented a fourth type of bitmapped resource. Application fonts, however, were limited to bitmapped font images. These custom fonts are no longer supported as resource elements.

Today, with a few exceptions, bitmapped fonts have been replaced by vectored (a.k.a. True Type) character fonts. These fonts have several advantages: They are resizable, adaptable to different screen resolutions, and generally cleaner and easier to read, as well as more attractive.

Vectored fonts, however, are not application resources; that is, they are not included as an integral part of the application. Instead, vectored fonts are used as system resources, available to all applications. These types of fonts are not intended to be application-specific.

To create custom fonts, consider using any of the numerous font editors available on the market (such as Fontographer or Adobe Font Manager). However, requiring a custom font or fonts for your application is not a recommended practice and should be done only under special circumstances.

A Bitmap Editor

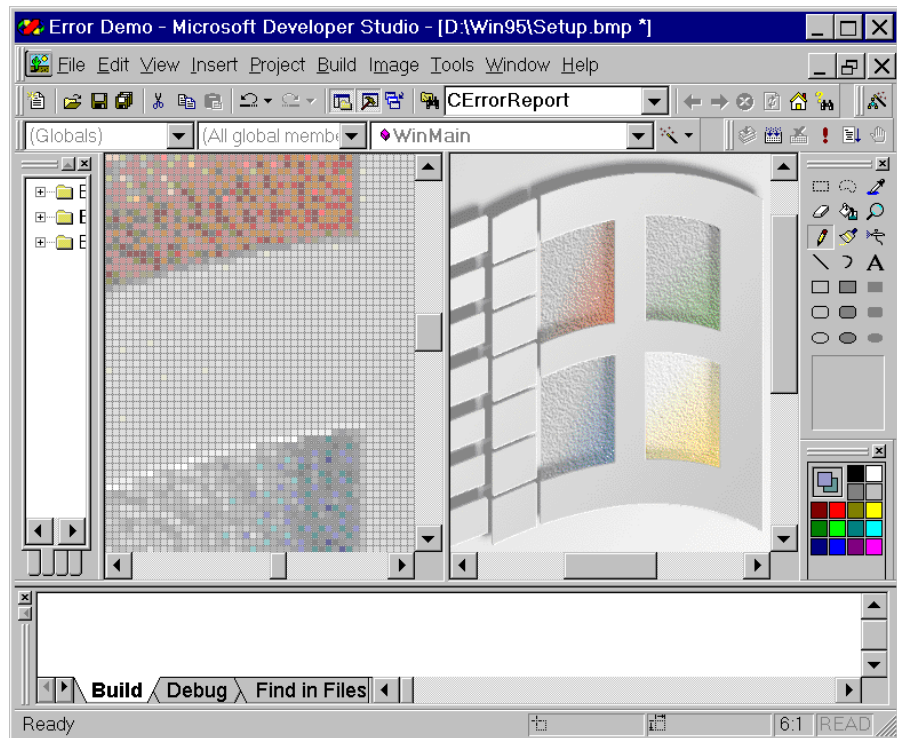
Only the icon and cursor image types require the specialized formats provided by an image editor. You can also create bitmaps using a wide variety of other paint utilities (many of which are better suited for general illustration than either of these resource editors).

Microsoft's image editor, which is part of the Microsoft Developer Studio, supports bitmap, cursor, icon, and toolbar images. The image editor is shown in Figure S6.1, where the Setup.BMP image (from the D:\Win98 directory) is in a split-window display with a zoom view on the left.

The palette bar appears (in the lower-right in Figure S6.1) with 16 basic colors. The foreground and background colors are shown at the upper left of the palette bar. Click (with the primary mouse button) to select a foreground color; right-click (with the secondary mouse button) to select a background color. To change any palette entry, double-click that entry to bring up the Windows common dialog box for color selection.

FIGURE S6.1:

The Microsoft Developer Studio image editor has a split-window display, with independent zoom capabilities.



The toolbar appears as a vertical, three-column bar (in the upper-right in Figure S6.1), with 21 tools. When you select a brush, pen, airbrush, or similar tool, the rectangle below the tool buttons offers a choice of weights or tool shapes for the selection.

The drawing operations are essentially the same as in other familiar paint programs, such as the Paint program distributed with Windows.

For more complex bitmap images, a wide variety of paint programs are available, and any .BMP or .DIB image can be imported as an application resource.

TIP

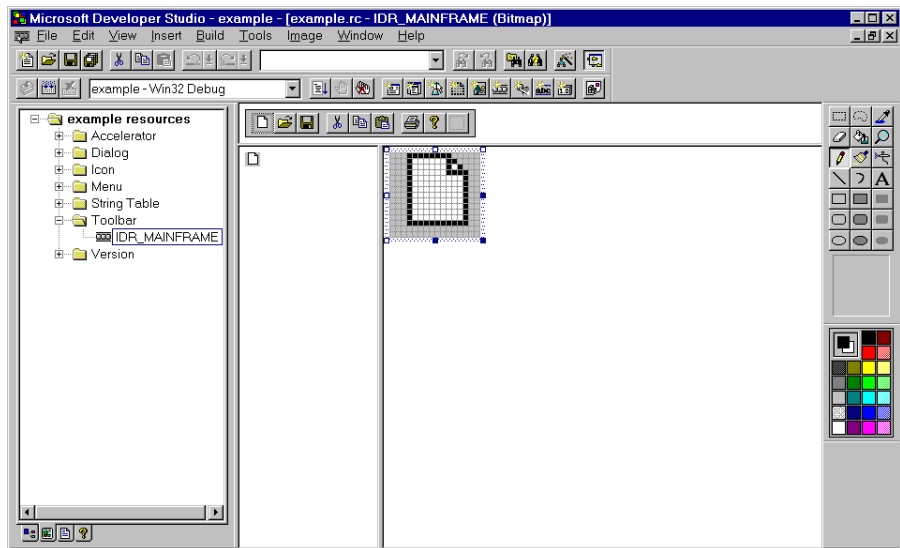
The tools in Microsoft's image editor should be familiar to anyone who has used a paint program. For explanations about how the individual tools function, use the editor's Help menu options.

Toolbar Resources

Toolbar images are stored as simple .BMP image files, but they have some additional information in the bitmap header to specify the number and width of the buttons and other organizational details. When a resource editor opens a toolbar bitmap, the strip image is displayed as buttons, with separators where appropriate, as shown in Figure S6.2.

FIGURE S6.2:

A toolbar image



The toolbar also shows two views of a selected button: one (on the left) in actual size and the second (on the right) enlarged for editing. Drawing operations for a toolbar button are essentially the same as for any bitmap image.

Unlike some bitmaps, you can enlarge toolbar images (to add new buttons) by selecting the blank button at the end of the toolbar image and then drawing in the button image. A new, blank button will be added to the toolbar strip automatically. Note, however, that the blank button is not stored as part of the toolbar bitmap and does not appear during use.

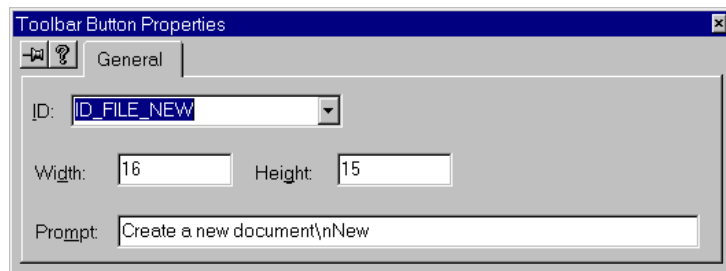
You create the separations between buttons by selecting a button on the button strip and dragging it to the right. When you release the button, a separator space appears in the button order. To remove a separator, simply reverse the process.

By dragging on any of the three black handles in a toolbar button (bottom-center, bottom-right, or right-center), you can resize the button using the mouse. When you change the size of existing toolbar buttons, the existing images are repositioned (and resized as necessary) to fit the new button size. In other words, if you enlarge the buttons, the existing images are repositioned (centered) within the new button area. If you reduce the buttons in size, the images may be truncated from the sides. (As buttons are reduced, the editor attempts to keep the images for each by trimming pixels.)

You can summon the Toolbar Button Properties dialog box for an individual button by double-clicking the button in the toolbar image (at the top of the editor window). Figure S6.3 shows the Toolbar Button Properties dialog box. Using this dialog box, you can edit the button ID (a default button ID is assigned when the button is created), change the width and height for the button, and enter the prompt strings for the button.

FIGURE S6.3:

The Toolbar Button Properties dialog box



The Prompt field consists of two strings separated by a *newline* (\n) character. The first string is the prompt that appears on the application's status bar (normally at the bottom of the application window). The second string entry provides the pop-up tip (normally brief) that appears when the cursor is positioned on the button.

TIP

Although there are no technical limits on the number of buttons a toolbar can have, you should remember that application window sizes might be limited by the size and resolution of the end users' display monitors. Instead of creating exceptionally long toolbars, it may be a better idea to break a long toolbar (more than a dozen buttons) into two smaller toolbars.

Icon Resources

Icon resources are used to represent applications, system resources, subsystems, or controls within an application. Icons are commonly used to represent available programs or minimized programs. Some applications, such as the Clock program distributed with Windows 98, create their own dynamic icons. Aside from such custom icons, conventional icons are simply small bitmaps of a symbol representing the application.

Designing an icon is largely a matter of personal taste. Icons can be as colorful and gaudy or as starkly plain as you choose. However, there are a few points you should keep in mind when designing icons:

- Your application is not going to stand or fall on the quality of your icon (unless you have a very unusual application).
- There is always a temptation to put a great deal of detail into an icon, or to reproduce a company logo, or to execute some concept that sounds fantastic in conversation. But remember, an icon is small, and much of the detail is simply going to be lost as clutter. In general, a simple icon is best. Its only real purpose is to be recognizable, so that the user can easily locate and select the icon to launch the application it represents.
- Although you can design icons using fine color details, if they are executed on a simple VGA system (or on many laptops), they will be mapped to the nearest available colors in the palette. The result is that all of your careful work, and quite likely the image as well, is lost. On a monochrome system, the results can be even worse when the colors are dithered down to black and white.

The best rule is simple: as in “keep it simple.” If you need contrast, you can always use inverted pixels to ensure that at least some of the icon will be visible regardless of the background.

Cursor Resources

Cursor resources (mouse pointers) are a specialized form of bitmap. Unlike other bitmap images, the bitmap images used for cursors do not replace the underlying

background images. Instead, cursor images are intended to interact with the background but leave the underlying image unchanged after the cursor moves.

For maximum visibility, cursor images are normally created as outlines using the inverted pixels for contrast or highlighting. The transparent pixels are simply used for areas where the background is allowed to show through.

As with previous Windows versions, only black-and-white cursor images are supported, and cursor pixels are composed of four colors: black, white, transparent, and inverted. (Animated cursors—which do permit color—are not a standard cursor image.)

Cursor Elements

Cursors provide three principal elements:

Pointer image The cursor image shows the mouse location and also frequently indicates the general function currently being executed.

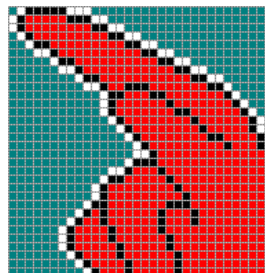
Screen image The screen image is a mask governing the interaction between the cursor (pointer) image and the underlying screen image. This image is generated automatically to fit the pointer image.

Hotspot The hotspot is a location within the cursor image that corresponds to the mouse's absolute screen position and is assigned by the cursor's designer.

Figure S6.4 shows a simple hand cursor with a pointing finger. The hand is drawn in black, then outlined in white, and finally, filled using *inverted* pixels. All these elements were designed to maximize the appearance of the image against any background—light, dark, or mixed. The hotspot is located at the tip of the extended finger.

FIGURE S6.4:

A sample cursor image



TIP

A simpler version of the cursor shown in Figure S6.4 could be drawn by using only inverted pixels for the image and leaving the rest of the field transparent. Alternatively, you could draw the image in black and use an inverted outline.

Animated Cursors

Animated cursors are a specialized format using the .ANI extension. They consist of a series of images that provide simple visual animation. A few interesting examples are distributed with Windows 98, including sand pouring through a small hourglass (AppStart.ANI), a spinning globe (Globe.ANI), and a large, animated hourglass (Hourglas.ANI). Animated cursors may also include color, unlike static cursor images.

The Microsoft C++ compiler suite does not provide support for creating or editing animated cursors, nor can animated cursors be readily imported as application resources. There are, however, third-party utilities available for animated cursor creation, if you are so inclined. Alternatively, you might create your own animation by using a separate thread to load a series of cursor images, which is essentially what an animated cursor does.

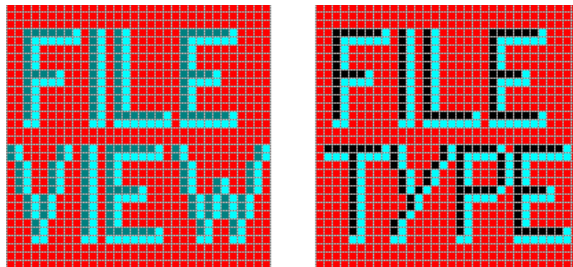
Two Icons for FileView



The *FileView* demo, which is described in detail in Supplement 10, uses two icon images, as illustrated in Figure S6.5. The primary application icon appears at the right; the icon on the left is used in one of the dialog boxes.

FIGURE S6.5:

Two resources for the
FileView demo



Both icon images were designed to stand out against any background, beginning with a field drawn using the inverse brush. The letters are drawn using the screen (transparent) brush, and then they are outlined in light cyan.

NOTE

The *FileView* demo (both a *FileView1* and a *FileView2* version) is included on the CD in the Supplement 10 folder. Additional resources used by the *FileView* demo are discussed in the following chapters.

In this chapter, you learned about the various types of image resources: bitmap, toolbar, icons, and cursor. These are easy to produce and edit using image editors. In the next chapter, we'll talk about dialog box resources.

Dialog Box Resources

- Dialog box editor features
- Dialog box properties
- Dialog box controls
- Dialog box alignment, positioning, and sizing

Dialog boxes are integral to Windows applications. Because many applications use dozens of—or even more—dialog boxes, the total number of dialog box resources easily exceeds all other resource elements.

Although you can use ASCII scripts to define dialog boxes, dialog box editors provide a much more convenient method. In this chapter, we'll cover the use of a dialog box editor and describe the various types of dialog box controls.

A Dialog Box Editor

Microsoft's dialog box editor, part of the Microsoft Developer Studio, provides interactive dialog box design. This editor offers tools to create and arrange all standard dialog box resource elements, including control buttons, checkboxes, edit boxes, list boxes, and radio buttons. The use of the dialog box editor is largely intuitive; in many respects, it operates much like a paint program, using drag-and-drop tools to position and size resource elements selected from a toolbar.

NOTE

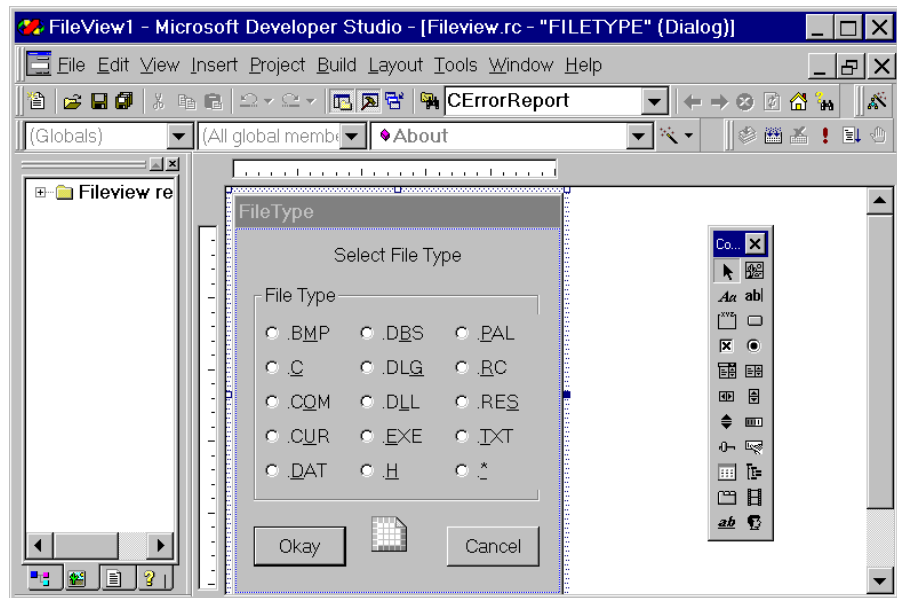
Borland's C++ Builder editor takes a different approach to constructing dialog boxes, using a format reminiscent of Visual Basic's "form-centric" theme. This is in contrast to Visual C++'s application-centric theme. C++ Builder even uses the term *form* rather than *dialog box*. However, even though they're called forms, the elements and construction are essentially the same as for building dialog boxes with Microsoft's Developer Studio or Borland's earlier Resource Workshop.

The main screen of the Microsoft Developer Studio dialog box editor is shown in Figure S7.1. Here you see a blank dialog box that contains only two buttons, the Controls toolbar (far right), and the status bar (bottom) with additional operator options and alignment tools.

The dialog box editor's Controls toolbar offers 22 tools: one represents the control cursor, 20 represent the standard dialog box element types, and the last (bottom-right) provides for custom dialog box resource types. These resource controls are discussed in more detail later in the chapter, in the section "Dialog Box Control Elements" (and illustrated later in Figure S7.7).

FIGURE S7.1:

The Microsoft Developer Studio dialog box editor



The Layout menu offers alignment and layout options, which are also available from the bottom toolbar. However, the menu also offers options for guide settings and tab-order arrangements, which are not found on the toolbar.

NOTE

The examples in this book were created using the Microsoft Developer Studio and Visual C++. However, the principles and designs are compatible with other compilers and tool sets, and in most cases, can be used with other languages as well.

Dialog Box Properties

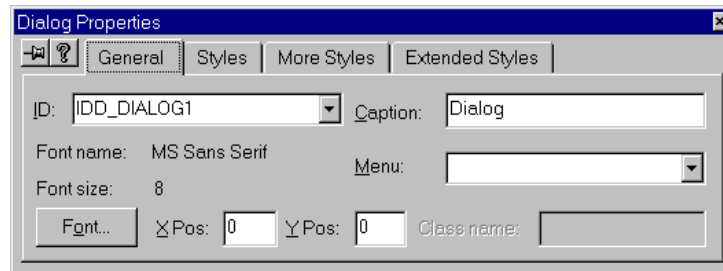
With a dialog box editor, you can create a variety of dialog box types and styles. To set the properties for a dialog box with Microsoft's dialog box editor, select Properties from the Edit menu, or right-click on the dialog box in the editor. This brings up the Dialog Properties dialog box, which has General, Styles, More Styles, and Extended Styles tabs.

General Properties

The General tab of the Dialog Properties dialog box, shown in Figure S7.2, includes the dialog box ID and caption, along with font information, position information, the dialog menu, and an associated class name.

FIGURE S7.2:

The General tab of the Developer Studio Dialog Properties dialog box



The General tab includes these fields:

ID A mnemonic symbol defined in the header file. This may be a symbol (the customary default), an integer, or a quoted string.

Caption Text appearing as the dialog box label. Change the default dialog box name supplied by the resource editor to a label identifying the purpose or function of the dialog box.

Menu An optional resource identifier for a menu to be used in the dialog box.

Font Name The typeface of the font used in all the controls in the dialog box. The bold version of the typeface is always used.

Font Size The point size for the font used in all the controls in the dialog box.

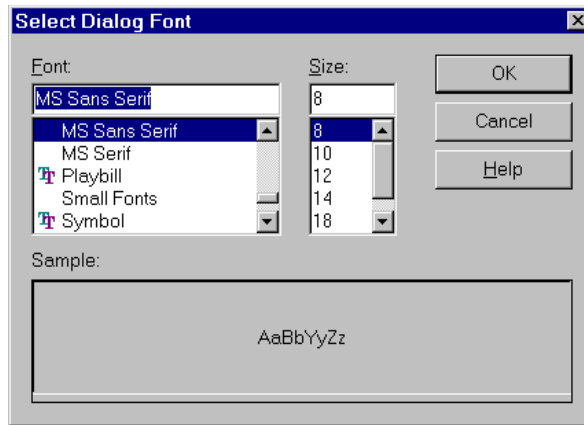
X Pos and Y Pos The x- and y- coordinates, in dialog box units (*DLUs*, a.k.a. dialog logic units), for the upper-left corner of the dialog box.

Class Name The registered dialog class (a Windows operating-system window class, not a C++ class). Provided to support C programming, this element is disabled when using MFC library support.

The General tab also contains a Font button, which calls the Select Dialog Font dialog box, as shown in Figure S7.3. Here, you can change the default typeface or point size used with your dialog box. A sample of the selected typeface and size is shown at the bottom of the Select Dialog Font dialog box.

FIGURE S7.3:

Choosing a default font for your dialog box

**NOTE**

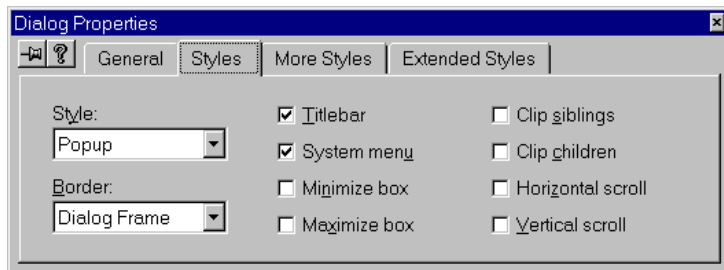
For applications written using Microsoft Visual C++, you may select only one font, which will be applied to all controls and text elements in the dialog box.

Dialog Box Styles

The Styles tab of the Dialog Properties dialog box offers controls for the overall appearance and behavior of the dialog box. Figure S7.4 shows this tab.

FIGURE S7.4:

The Styles tab of the Developer Studio Dialog Properties dialog box



Dialog box styles may have the following values:

Style You can choose from the following operation styles:

- Overlapped, which is always a top-level window and should have a caption and a border. Overlapped windows are pop-up windows that

can be overlapped by other dialog box windows. Normally, only the main window in an application is defined as Overlapped.

- **Popup**, which is the default. Pop-up dialog boxes appear only when called by an application in response to a menu selection or some other program instruction.
- **Child**, which creates a dialog box defined as a child window belonging to another window. Child dialog boxes are generally used when several tiled windows are desired within an application, and they are displayed at all times (unless, of course, they are covered by another window or application). Child windows belonging to an application are not allowed to overlap.

Border You can choose from four frame (border) styles, which determine the appearance of the dialog box frame and the presence or absence of a caption bar:

- **None**, which displays neither a border nor a caption bar.
- **Thin**, which displays a thin, single border without a caption bar.
- **Resizing**, which displays a double border without a caption bar.
- **Dialog Frame**, the default, which displays a double border with a caption bar.

Titlebar If checked (the default), the dialog box appears with a title bar.

System Menu If checked (the default), the dialog box appears with a system menu at the upper-left corner of the frame. The system menu appears only on captioned dialog boxes.

Minimize Box If checked (unchecked is the default), the dialog box appears with a Minimize box at the upper-right corner of the frame. The Minimize box appears only on captioned dialog boxes.

Maximize Box If checked (unchecked is the default), the dialog box appears with a Maximize box at the upper-right corner of the frame. Like the system menu and the Minimize box, the Maximize box appears only on captioned dialog boxes.

Clip Siblings If checked (unchecked is the default), child windows are clipped relative to each other. Thus, when a particular child window is repainted, all other top-level child windows are clipped from the region

of the child window to be updated. If cleared and child windows overlap, drawing in the client area of a child window may draw in the client area of a neighboring child window. This option is for use with child windows only.

Clip Children If checked (unchecked is the default), this excludes the area occupied by child windows when drawing within the parent window. This option is used only when creating a parent window. Do not use this style if the dialog box contains a group box.

Horizontal Scroll If checked (unchecked is the default), the dialog box contains a horizontal scrollbar.

Vertical Scroll If checked (unchecked is the default), the dialog box contains a vertical scrollbar.

WARNING

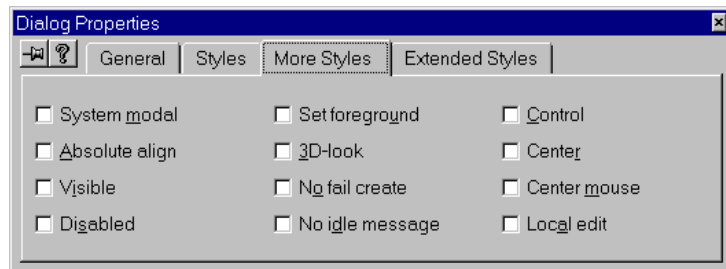
When a dialog box using the default Border style (Dialog Frame) contains scrollbars, the scrollbars are drawn overlapping the borders of the dialog box rather than inside the frame, and the contents of the dialog box may be clipped incorrectly. This is standard Windows behavior. Therefore, to use scrollbars with a dialog box frame, select None, Thin, or Resizing for the Border style. This does not apply to scrollbar controls within the dialog box—only to scrollbars used to scroll the dialog box itself.

More Dialog Box Style Options

The More Styles tab of the Dialog Properties dialog box offers even more appearance and behavioral controls for the dialog box. Figure S7.5 shows this tab.

FIGURE S7.5:

The More Styles tab of the Developer Studio Dialog Properties dialog box



This tab offers the following selections, which are all unchecked by default:

System Modal Makes the dialog box system-modal, which prohibits switching to another window or program while the dialog box is active. This option is used for warnings, queries, and other urgent or immediate messages.

Absolute Align Aligns the dialog box relative to the upper-left corner of the screen. (By default, the dialog box is aligned relative to its parent window.)

Visible Makes the dialog box visible when first displayed. This option is applicable to overlapping and pop-up windows. Do not check this option for form views and dialog-box template resources.

Disabled Disables the dialog box when it first appears.

Set Foreground Brings the dialog box to the foreground by internally calling the `SetForegroundWindow` function for the dialog box.

3D-look Makes the dialog box appear with a nonbold font and draws three-dimensional borders around control windows in the dialog box.

No Fail Create Creates the dialog box even if errors occur. For example, if a child window cannot be created or if the system cannot create a special data segment for an edit control, the dialog box will still be created.

No Idle Message Suppresses the `WM_ENTERIDLE` message ordinarily sent to a dialog box's owner when no more messages are waiting in its message queue. This option is valid only for modal dialog boxes.

Control Creates a dialog box that works well as a child window of another dialog box, similar to a page in a property sheet. This option permits the user to tab among the control windows of a child dialog box, use its accelerator keys, and so on.

Center Centers the dialog box in the working area; that is, the area not obscured by the toolbar.

Center Mouse Centers the mouse cursor in the dialog box on opening.

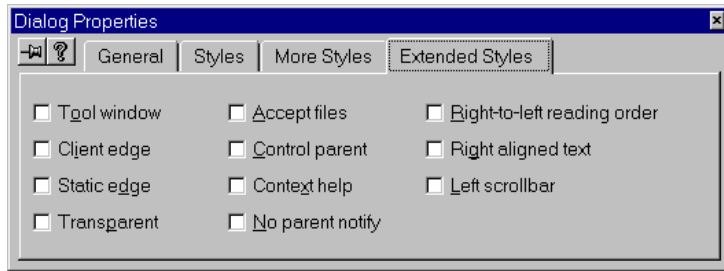
Local Edit Specifies that edit-box controls in the dialog box will use memory in the application's data segment. Normally, all edit-box controls in dialog boxes use memory outside the application's data segment. This option should always be used if the application will be using the `EM_SETHANDLE` or `EM_GETHANDLE` messages.

Extended Dialog Box Style Options

The Extended Styles tab of the Dialog Properties dialog box offers additional appearance and behavioral controls for the dialog box. Figure S7.6 shows this tab.

FIGURE S7.6:

The Extended Styles tab of the Developer Studio Dialog Properties dialog box



This tab offers the following selections, which are all unchecked by default:

Tool Window Creates a tool window intended to be used as a floating toolbar. Tool windows have a shorter-than-normal title bar, and the title is drawn using a smaller font.

Client Edge Creates a border with a sunken edge around the dialog box.

Static Edge Creates a border around the dialog box.

Transparent Creates a dialog box with a transparent window, so any windows beneath the dialog window are not obscured. The dialog window receives `WM_PAINT` messages only after all sibling windows beneath it have been updated. This option is useful for overlay drawing, but it does not function well for overlaying live video.

Accept Files Allows the dialog box to accept drag-drop files. When a file is dropped on a dialog box, a `WM_DROPFILES` message is sent to the control.

Control Parent Allows the user to navigate among the dialog child windows using the Tab key. (Navigation using the mouse also remains in effect.)

Context Help Includes the Help question mark icon in the dialog box's title bar. When the user clicks the question mark, the cursor changes to a question mark with a pointer. Then, when he or she clicks a child window, the child receives a `WM_HELP` message. The `WM_HELP` message should be

passed to the parent window procedure, which should call the `WinHelp` function using the `HELP_WM_HELP` command, so that the Help application can display a pop-up window with help information for the child window.

No Parent Notify Stops the child window from sending the `WM_PARENTNOTIFY` message to its parent window.

Right-to-Left Reading Order Displays the dialog box text using right-to-left reading order properties.

Right Aligned Text Right-aligns text within the dialog box.

Left Scrollbar Displays the vertical scrollbar (if present) to the left of the client area.

Dialog Box Control Elements

Dialog boxes may contain a wide variety of controls, including buttons, scrollbars, list boxes, edit fields, images, spin buttons, and more. New control varieties are introduced regularly. While it would be literally impossible to describe every type of control, I will attempt to cover most of the standard types here, even though “new” standard types appear almost as often as new custom types.

The dialog box controls are on the dialog box editor’s Controls toolbar. Figure S7.7 shows the Microsoft Developer Studio Controls toolbar, labeled with the names of the toolbar buttons. To select these control types, click the appropriate button, and then position the control in the dialog box outline.

Button Types

Buttons are used to define controls that permit user interactions. Three types of dialog box buttons are provided: pushbuttons, checkboxes, and radio buttons.

Pushbuttons

Pushbuttons are the simplest form of dialog box control. They execute an immediate response when you click them with the mouse, but normally do not maintain any status information. Pushbuttons usually contain a text label (caption) identifying their purpose.

When you choose the Button button on the Controls toolbar, you'll see the dialog box shown in Figure S7.8.

FIGURE S7.7:

Use the Controls toolbar in the Microsoft Developer Studio dialog box editor to add dialog box controls.

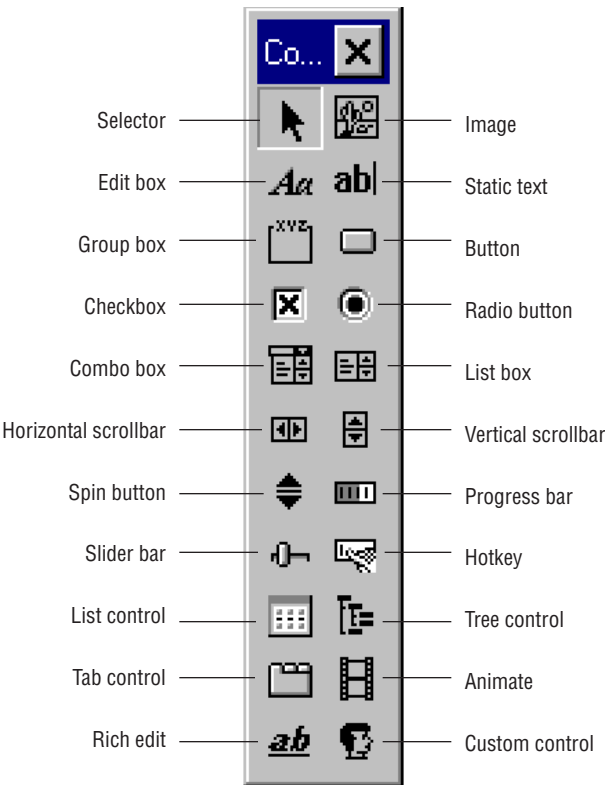
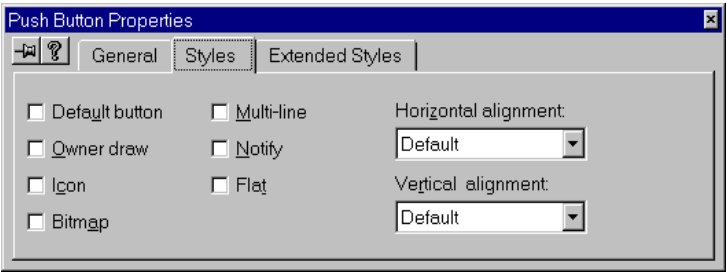


FIGURE S7.8:

Choosing pushbutton styles



The Push Button Properties dialog box offers the following styles:

Default Button Makes the control the default button in the dialog box. The default button is drawn with a heavy black border when the dialog box first appears, and it is executed if the user presses Enter without choosing another command in the dialog box. Windows allows only one default button in a dialog box.

WARNING

Unfortunately, the Developer Studio does not prevent you from creating more than one button with the Default Button style. The result of having multiple default buttons in a dialog box is generally failure of any of the buttons to respond to the Enter key.

Owner Draw Used when an application needs to customize the appearance of a control. When you select this style, Windows does not handle the button appearance. Instead, when the button is activated, the parent window is notified with a request to paint, invert, or disable the button. The application must provide its own `OnDrawItem` message handler in the owner-window procedure (either the dialog box procedure or class derived from MFC class `CDialog` or `CFormView`). Owner-draw classes may also be derived from `CButton` using an override for the `CButton::DrawItem` method.

Icon Displays an icon image for the button.

Bitmap Displays a bitmap image for the button.

Multi-line Allows the button text to wrap to multiple lines if it is too long to fit on a single line within the button rectangle.

Notify Sends a notification to the parent window (the dialog box) when the pushbutton is clicked or double-clicked. By default, this option is not selected and the button functions by generating a message—using the button ID—when selected.

Flat Creates a flat button without three-dimensional shading.

Horizontal Alignment Offers a choice of how the control's caption text is positioned horizontally. Options are Default (centered), Left, Center, or Right.

Vertical Alignment Offers a choice of how the control's caption text is positioned vertically. Options are Default (centered), Top, Center, or Bottom.

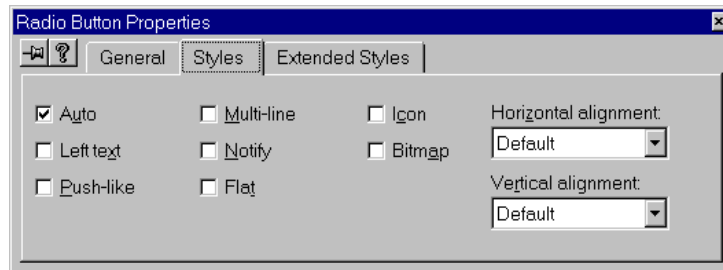
Radio Buttons

Radio buttons and auto radio buttons are used to make a selection from a list of mutually exclusive options. Usually, when radio buttons are toggled on, dots appear inside the button. Labels adjacent to the buttons identify the option or selection. By convention, you can only select one button at a time in any group; all other buttons in the same group should be cleared.

Regular radio buttons require provisions within the application to send a set/clear message back to the button to initiate a change of state. Auto radio buttons, which look just like regular radio buttons, automatically reset their own state and, when selected, also clear any other radio buttons belonging to the same group. Whether or not radio buttons or auto radio buttons are used, the application is responsible for setting the initial, default selection in each group when the dialog box is initiated.

When you choose the Radio Button button on the Controls toolbar, you'll see the dialog box shown in Figure S7.9.

FIGURE S7.9:
Choosing radio button styles



The Radio Button Properties dialog box offers the following styles:

Auto Displays the checked state automatically when the user selects the radio button. At the same time, any other radio buttons in the group are cleared (deselected). When a group of radio buttons is used with the Dialog Data Exchange (DDE), the Auto property must be set. This style is checked by default.

NOTE

The Dialog Data Exchange (DDE) is an MFC-based mechanism that provides the exchange of data between dialog box elements and corresponding member variables within the `CDialog`-derived class. Using the DDE, instead of explicitly generating messages to read or write the state, text, or other values from and to dialog box elements, calling the `UpdateData` function with a `FALSE` argument causes all the elements to be updated with the values from their corresponding class members. Calling `UpdateData` with a `TRUE` argument retrieves the current values from all dialog box elements, storing (and validating when appropriate) these values in the member variables.

Left Text Places the radio button's caption text on the left of the button rather than the right.

Push-like Gives the radio button the appearance of a conventional push-button while still retaining the performance and characteristics of a radio button. A push-like radio button appears raised when unchecked and sunken when checked (pushed).

Multi-line Allows the radio button text to wrap to multiple lines if it is too long to fit on a single line within the button rectangle.

Notify Notifies the parent window when the radio button is clicked or double-clicked. Notification is used only when the parent is expected to take immediate action in response to a change rather than waiting to query the button status when the dialog box session concludes.

TIP

Because radio buttons are customarily used to establish settings or selections that will only become relevant after the dialog box closes—when the application returns to its primary tasks—the normal expectation is that the status of a radio button is queried only after the dialog box closes. This is not, however, a hard and fast rule; it's merely a generality. When an immediate response to a change in state is required, select the `Notify` option.

Flat Creates a flat radio button without three-dimensional shading.

Icon Displays an icon image for the radio button.

Bitmap Displays a bitmap image for the radio button.

Horizontal Alignment Offers a choice of how the control's caption text is positioned horizontally. Options are `Default` (text to the right of the button), `Left`, `Center`, or `Right`.

Vertical Alignment Offers a choice of how the control's caption text is positioned. Options are Default (centered), Top, Center, or Bottom.

Checkboxes

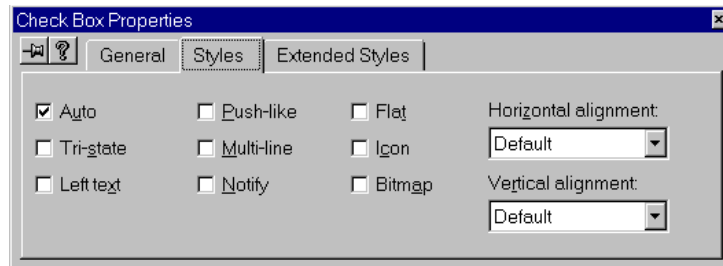
Like radio buttons, checkboxes are usually identified by text labels set to the right or left of the checkbox image. Unlike radio buttons, checkboxes permit selection of none, one, or multiple items. Each checkbox resets its own image by displaying a checkmark when selected. A second mouse click on a checkbox cancels selection, resetting the image.

Checkboxes may be grouped, but they do not interact with others in a group. Each checkbox selection is assumed to be made independently of any other selections.

When you choose the Checkbox button on the Controls toolbar, you'll see the dialog box shown in Figure S7.10.

FIGURE S7.10:

Choosing checkbox styles



The Check Box Properties dialog box offers the following styles:

Auto Toggles between the checked and unchecked states automatically when the user selects the checkbox. When checkboxes are used with the DDE, this property must be set to TRUE. This style is checked by default.

Tri-state Allows the checkbox to display three states: checked, cleared, or grayed. A grayed checkbox indicates that the state represented by the control is undetermined or, alternately, the state of the checkbox selection is irrelevant (disabled).

Left Text Places the checkbox's caption text on the left of the checkbox rather than the right.

Push-like Gives the checkbox the appearance of a conventional push-button while still retaining the performance and characteristics of a checkbox. A push-like checkbox appears raised when unchecked and sunken when checked (pushed). For a tri-state push-like checkbox, the third state is depressed but grayed.

Multi-line Allows the button text to wrap to multiple lines if it is too long to fit on a single line within the button rectangle.

Notify Notifies the parent window when a checkbox is clicked or double-clicked. Notification is used only when the parent is expected to take immediate action in response to a change rather than waiting to query the button status when the dialog box session concludes.

Flat Creates a flat checkbox without three-dimensional shading.

Icon Displays an icon image for the checkbox.

Bitmap Displays a bitmap image for the checkbox.

Horizontal Alignment Offers a choice of how the control's caption text is positioned. Options are Default (left), Left, Center, or Right.

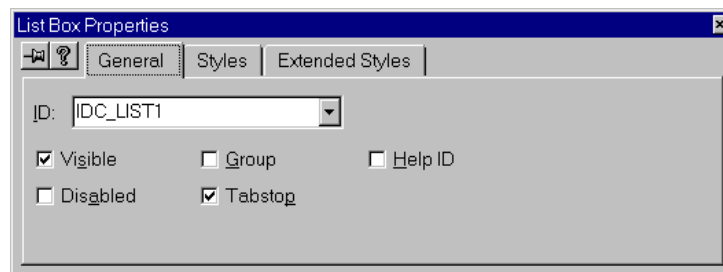
Vertical Alignment Offers a choice of how the control's caption text is positioned. Options are Default (centered), Top, Center, or Bottom.

General Properties for Other Controls

The General tab of the Properties dialog boxes for the text-oriented fields, range and adjustment controls, and other controls is similar for each type. Figure S7.11 shows the General tab of the Text Properties dialog box.

FIGURE S7.11:

The General tab of the Text Properties dialog box



The text-oriented, range and adjustment, and other controls have the following general properties in common:

ID Indicates the default resource ID for the control. You may retain the supplied ID or change to a mnemonic ID by entering the desired identifier. When you supply a new ID, a value is assigned automatically, and an entry is placed in the `Resource.H` header. The resource ID may be a symbol, integer, or quoted string.

Visible Makes the control visible when the application is first run. This option is checked by default.

Disabled Displays the resource as disabled when the dialog box is created (not relevant to static text controls).

Group Makes the control the first control of a group of controls, where users can move from one control to the next by using the arrow keys. All controls in the tab order after the first control belong to the same group if the Group property is set to `FALSE` (unchecked). The next control in the tab order that has Group set to `TRUE` (checked) ends the first group of controls and starts the next group. For static text fields, this option is checked by default.

Tabstop Allows the user to move to this control with the Tab key. This option is checked by default for all text-oriented controls except static text fields. It is also the default for sliders, hotkeys, and animated controls.

Help ID Assigns a help ID to the control based on the resource ID (not relevant to static text).

Text-Oriented Fields

Six text-field types are provided: static text fields, edit boxes, list boxes, combo boxes, list control boxes, and tree controls.

Static Text Fields

Static text fields display labels and other information that cannot be entered or changed by the user. They may show information, ask questions, provide explanations, or simply provide labels for other controls or for edit boxes. Static text fields may be formatted as left-justified, right-justified, or centered.

When a static text field is created, a default identifier, `IDC_STATIC`, is assigned and the default caption *Static* is supplied. If an application needs to change the contents for a static text display, an individual resource ID should be assigned.

TIP

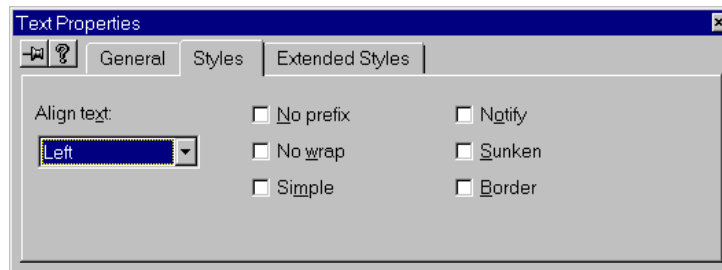
Static text fields may be assigned unique identifiers if there is any reason to have the displayed text change during execution of the application. After assigning a unique ID, the `SetDlgItemText` function can be used to assign new text to the static text field.

Along with the properties listed in the previous section, the General tab of the Text Properties dialog box contains a Caption property for the text string that appears in the static text field. If you want to change the default caption *Static*, this is where you would enter the new caption.

In addition to setting the options on the General tab for a static text field, you can also set the control's appearance using the Styles tab, shown in Figure S7.12.

FIGURE S7.12:

Setting static text field styles

**WARNING**

Static text fields are limited to 255 characters. Multiple static text fields may overlap and conceal portions of other fields.

The Styles tab of the Text Properties dialog box offers the following options:

Align Text Controls how text is aligned in the static text control: Left (the default), Center, or Right. This should be set to Left when No Wrap is selected.

No Prefix Prevents ampersands (&) in the control's text from being interpreted as the mnemonic character. Normally, a string containing an ampersand is displayed with the ampersand removed and the next character in the string underlined. The No Prefix style is most often used

when filenames or other strings that may contain an ampersand need to be displayed.

No Wrap Displays text left-aligned; tabs are expanded but text is not wrapped, and any text extending beyond the end of a line is clipped.

Simple Disables both the No Wrap and Align Text options; text does not wrap and is not clipped. Furthermore, overriding `WM_CTLCOLOR` in the parent window has no effect on the control.

Notify Notifies the parent window if the control is clicked or double-clicked (not applicable to static text).

Sunken Creates a border with a sunken edge around the static text control.

Border Creates a border around the text control.

Edit Boxes

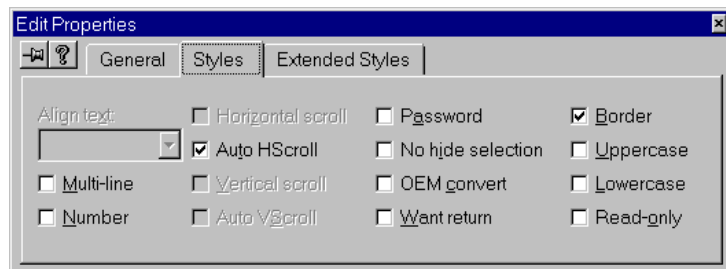
Edit boxes are used for entries and responses, but they may also display information or selections without allowing the user to change the entry. Usually, edit boxes permit the user to enter new text information or to edit existing text information. Although only one type of edit box is listed in the resource editor, these controls can be defined as single-line or multiline edit fields and may include vertical and/or horizontal scrolling and scrollbars.

When you create an edit box, the resource editor supplies a default ID value: `IDC_EDITn`. You can enter a new ID or select one from a list of defined IDs in the General tab of the Edit Properties dialog box. See the section “General Properties for Other Controls” earlier in the chapter for a description of the other properties on the General tab.

In addition to setting the properties on the General tab, you can also set the control’s appearance using the Styles tab, as shown in Figure S7.13.

FIGURE S7.13:

Setting edit box styles



The Styles tab of the Edit Properties dialog box has the following options:

Align Text Offers a choice of Left (the default), Centered, or Right-aligned text (when the Multi-line option is selected).

Multi-line Creates a multiline edit-box control. When a multiline edit box is in a dialog box, pressing the Enter key selects the default button. Multiline edit boxes may have scrollbars and process their own scrollbar messages. They may also process scrollbar messages sent by the parent window. See also Horizontal Scroll, Auto HScroll, Vertical Scroll, and Want Return.

Number Restricts input to numeric characters and associated symbols; prevents any nonnumeric characters from being typed.

Horizontal Scroll Adds a horizontal scrollbar to a multiline control. This option is not available unless the Multi-line option has been selected.

Auto HScroll Scrolls text right automatically when a character is typed at the right end of the box. This option is checked by default. When Auto HScroll is selected, text automatically scrolls horizontally whenever the caret (text cursor) passes the right edge of a multiline edit box. The user must press the Enter key to start a new line. If Auto HScroll is not selected, the control automatically wraps words to the beginning of the next line when necessary.

Vertical Scroll Adds a vertical scrollbar to a multiline edit-box control. This option is not available unless the Multi-line option has been selected.

Auto VScroll In a multiline edit box, automatically scrolls text up one line when the user presses Enter on the last line. This option is not available unless the Multi-line option has been selected.

Password Displays all characters typed as asterisks (*). This property is not available for multiline edit boxes.

No Hide Selection Controls how text is displayed when an edit box loses and regains the focus. If set, text remains selected even when the edit box loses the focus.

OEM Convert Converts text typed in the edit box from the Windows character set to the OEM character set and then back to the Windows set. Selecting this option ensures proper character conversion when the application calls the `AnsiToOem` function to convert a Windows string in the edit box to OEM characters. This property is most useful for edit-box controls containing filenames.

NOTE

OEM stands for original equipment manufacturer, but it refers to any kind of third-party addition, including special character sets for international use.

Want Return Inserts a carriage return when the user presses the Enter key while typing text in a multiline edit box. If Want Return is not set, pressing the Enter key is the same as pressing the dialog box's default pushbutton. Want Return has no effect on single-line edit boxes.

Border Draws a border around the edit box. This option is checked by default.

Uppercase Converts all characters typed to uppercase.

Lowercase Converts all characters typed to lowercase.

Read-Only Prevents users from changing the contents of the edit box.

List Boxes

List boxes display text (or icon) lists, allowing the user to select one (or more) items. List box entries are supplied by the application. For example, a list box might contain a list of filenames. When the list is too long for the allocated space, a scrollbar appears for vertical scrolling.

Custom list boxes can also be defined as owner-drawn list boxes. Custom controls may include graphic as well as text entries, but they require provisions within the application to handle the display material. To create a custom list box, select either the Owner Draw Fixed or Owner Draw Variable option on the Styles tab of the List Box Properties dialog box. The Owner Draw Fixed style requires that all items in the list box have the same height; the Owner Draw Variable style permits the mixing of items of varying heights.

NOTE

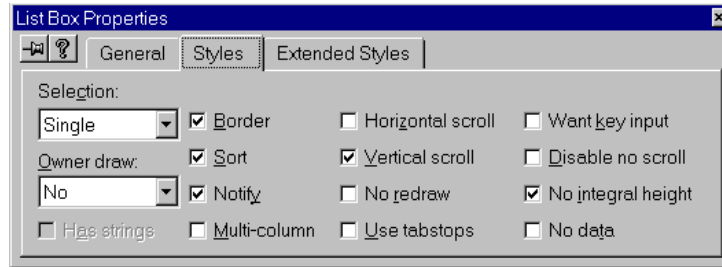
For more information about custom controls using the owner-drawn styles, refer to the Microsoft SDK.

When you create a list box, the resource editor supplies a default ID value: `IDC_LISTn`. You can enter a new ID or select one from a list of defined IDs in the General tab of the List Box Properties dialog box (see Figure S7.11, shown earlier). This dialog box contains the properties listed earlier in the section titled "General Properties for Other Controls."

The Styles tab of the List Box Properties dialog box contains options for setting the Control's appearance. Figure S7.14 shows this tab.

FIGURE S7.14:

Setting list box styles



The Styles tab contains the following properties for list boxes:

Selection Determines how items in a list box can be selected. Possible values are as follows:

- **Single**, which allows only one item in a list box to be selected at a time. This is the default selection method.
- **Multiple**, which allows more than one list-box item to be selected at a time. Clicking or double-clicking any unselected item selects it. Clicking or double-clicking any selected item deselects it. The Shift and Ctrl keys have no effect.
- **Extended**, which allows the Shift and Ctrl keys to be used together with the mouse to select and deselect list box items, select groups of items, and select nonadjacent items.

Owner Draw Sets the owner-draw characteristics for the list box using one of the following values:

- **No**, which turns off the owner-draw style, limiting the list box contents to strings. This is the default setting.
- **Fixed**, which makes the owner of the list box responsible for drawing the contents of the list box. All items in the list box must be the same height. `CWnd::OnMeasureItem` is called when the list box is created, and `CWnd::OnDrawItem` is called when a visual aspect of the list box has changed.
- **Variable**, which makes the owner of the list box responsible for drawing the contents of the list box; however, items in the list box may be of

varying heights. `CWnd::OnMeasureItem` is called for each item in the list when the list box is created, and `CWnd::OnDrawItem` is called when a visual aspect of the list box has changed.

Has Strings Specifies that an owner-drawn list box contains string items. The list box maintains the memory and pointers for the strings, allowing the application to use the `LB_GETTEXT` message to retrieve the text for a particular item. This option is available only if the Owner Draw option is set to either Fixed or Variable. If Owner Draw is set to No, the list box contains strings by default.

Border Creates a border around the list box. This option is checked by default.

Sort Sorts the contents of the list box alphabetically. This option is checked by default.

Notify Sends a notification to the parent window when the user clicks or double-clicks a list box item. This option is checked by default.

Multi-column Creates a multiple-column list box. In a multicolumn list box, the user scrolls horizontally. Use the `LB_SETCOLUMNWIDTH` message to set the width of the columns.

Horizontal Scroll Creates a horizontal scrollbar for the list box.

Vertical Scroll Creates a vertical scrollbar for the list box. This option is checked by default.

No Redraw Specifies that a list box's appearance is not updated when changes are made. You can change the No Redraw style via a `WM_SETREDRAW` message or by calling `CWnd::SetRedraw`.

Use Tabstops Allows a list box to recognize and expand tab characters when drawing its strings. The default tab positions are 32 dialog box units (DLUs).

Want Key Input Sends the list box owner `WM_VKEYTOITEM` messages when the Has Strings style is used or `WM_CHARTOITEM` messages whenever a key is pressed and the list box has the input focus. The Want Key Input option allows an application to perform special processing on the keyboard input.

Disable No Scroll Displays a disabled vertical scrollbar in the list box when there are not enough items to scroll. By default, a scrollbar does not appear until the list box contains enough items to scroll.

No Integral Height Specifies that the size of the list box is exactly the size specified by the application when the list box was created. Normally, the list box is sized so partial items are not displayed. This option is checked by default.

No Data Prevents the list box from storing item data.

Combo Boxes

Combo boxes combine the features of edit boxes and list boxes. They permit the user to either select from a list or type an entry directly in the edit box.

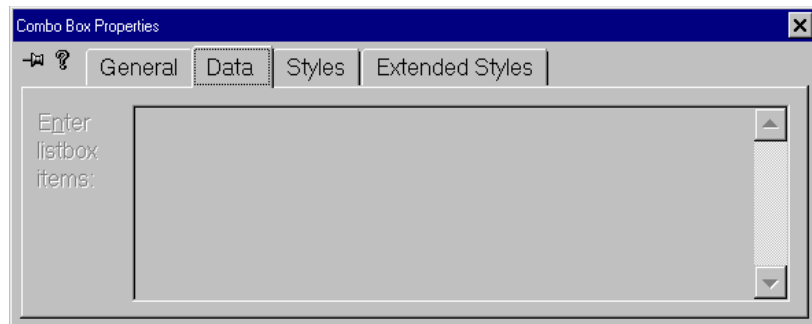
Three styles of combo boxes are supported: simple, drop-down, and drop-down-list combo boxes (use the Styles tab of the Combo Box Properties dialog box to select these options). Custom combo boxes are defined using the Owner-Draw option on the Styles tab as well. These options will be covered in more depth in a moment.

As with the other resource types, the resource editor supplies a default ID value, `IDC_COMBO0n`, when you create the combo box, but you can enter or select a new ID in the General tab of the Combo Box Properties dialog box. This dialog box contains the properties listed earlier in the section titled “General Properties for Other Controls.”

The Data tab of the Combo Box Properties dialog offers one additional property, as shown in Figure S7.15. The Enter Listbox Items property is available only in resource files using MFC library support. This property allows you to enter the initial selections that will appear in the list portion of the combo box when the dialog box is created. To add entries, press Ctrl+Enter (line feed) at the end of each item to move to the next line.

FIGURE S7.15:

Setting the list box contents for combo boxes



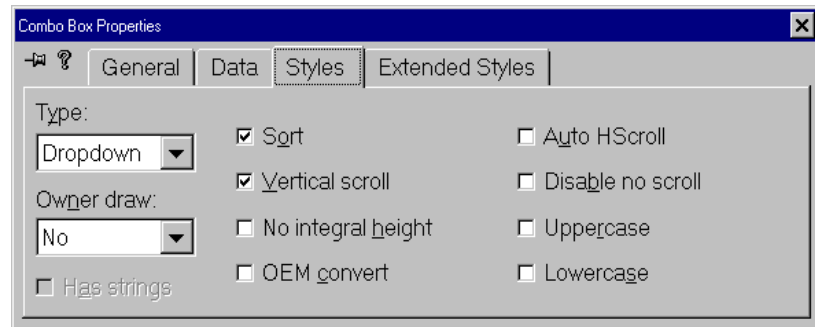
WARNING

A bug in Developer Studio 97 (Visual C++ version 5.0) causes the list box entry field in the Data tab to be permanently disabled. Earlier versions of VC++ offered this same functionality in a slightly different format without problems. At the time of this writing, there is no known patch to fix this problem, but you can edit the resource script directly—as ASCII text—to enter a default string list. Of course, irrespective of the resource script, an application can always add string entries at runtime.

The Styles tab of the Combo Box Properties dialog box allows you to set properties that affect the combo box's appearance. Figure S7.16 shows this tab.

FIGURE S7.16:

Setting combo box styles



The Styles tab contains the following properties for combo boxes:

Type Specifies the combo box as one of the following types:

- Simple, which creates a simple combo box combining an edit-box control for user input with a list-box control. The list is visible at all times, with the current selection from the list displayed in the edit-box control.
- Dropdown, which creates a drop-down combo box. A drop-down combo box is the same as a simple combo box, except the list is displayed only when the user selects the drop-down arrow at the right of the edit-box control portion. Because the list appears only on demand, the area used can also be occupied by other controls. This is the default type.

- **Drop List**, which creates a drop-down-list combo box. A drop-down-list combo box is similar to the drop-down combo box except that the edit-box control is replaced by a static-text item displaying the current list selection. It does not accept user input; edit field entries must correspond to items already in the list.

Owner Draw Sets the owner-draw characteristics for the combo box using one of the following values:

- **No**, which turns off the owner-draw style, limiting the list box contents to strings. This is the default setting.
- **Fixed**, which makes the owner of the combo box responsible for drawing the contents of the list box. All items in the list box must be the same height. `CWnd::OnMeasureItem` is called when the list box is created, and `CWnd::OnDrawItem` is called when a visual aspect of the list box has changed.
- **Variable**, which makes the owner of the combo box responsible for drawing the contents of the list box; however, list box items may be of varying height. `CWnd::OnMeasureItem` is called for each item in the list when the list box is created, and `CWnd::OnDrawItem` is called when a visual aspect of the list box has changed.

Has Strings Specifies that an owner-drawn combo box contains string items. The list box maintains the memory and pointers for the strings, allowing the application to use the `LB_GETTEXT` message to retrieve the text for a particular item. This option is available only if the Owner Draw option is set to either Fixed or Variable. If Owner Draw is set to No, the list box contains strings by default.

Sort Sorts the contents of the combo box alphabetically. This option is checked by default.

Vertical Scroll Creates a vertical scrollbar for the list box. This option is checked by default.

No Integral Height Specifies that the size of the combo box is exactly the size specified by the application when the combo box was created. Normally, the combo box is sized so partial items are not displayed.

OEM Convert Converts text typed in the combo-box control from the Windows character set to the OEM character set and then back to the Windows set. This option ensures proper character conversion when the

application calls the `AnsiToOem` function to convert a Windows string in the combo box to OEM characters. OEM Convert is most useful for combo boxes that contain filenames.

Auto HScroll Scrolls text right automatically when the user types a character at the right end of the box.

Disable No Scroll Displays a disabled vertical scrollbar in the list box when there are not enough items to scroll. By default, a scrollbar does not appear until the list box contains enough items to scroll.

Uppercase Converts all characters typed to uppercase.

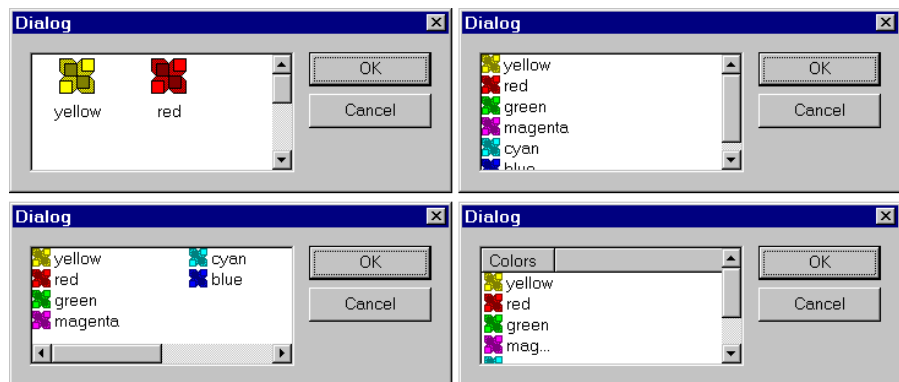
Lowercase Converts all characters typed to lowercase.

List Control Boxes

List control boxes are an extension of the list-box type, with the additional capabilities of displaying either large or small icons, a multicolumn list with icons, or in a report format, columnar lists with a header. The four styles of list control boxes are illustrated in Figure S7.17.

FIGURE S7.17:

Four list control box styles

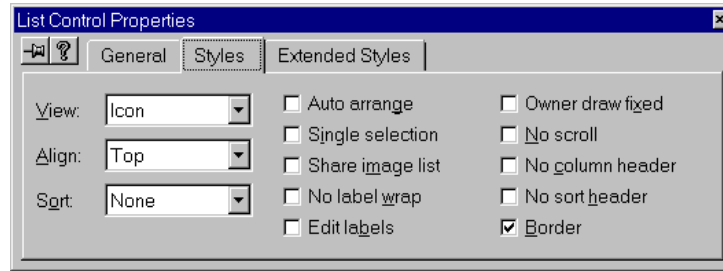


When a list control box is created, the default ID value `IDC_LISTn` (which follows the same format as a list box) is supplied. You can enter a new ID or select from a list of defined IDs in the General tab of the List Control Properties dialog box. This dialog box contains the properties listed in the earlier section titled “General Properties for Other Controls.”

You set the appearance of the list control box using the Styles tab of the List Control Properties dialog box, shown in Figure S7.18.

FIGURE S7.18:

Setting the styles of list controls



The Styles tab contains the following properties for list control boxes:

View Sets the display view for the list control box as one of the following:

- Icon, which sets the large icon view with the icons in a multicolumn arrangement. This is the default view.
- Small Icon, which sets the small icon view with the icons in a multicolumn arrangement.
- List, which sets a list view as a single-column display with small icons (optional) along the left.
- Report, which sets a report view for a multicolumn text display with a column header.

Align Sets the alignment of icons in the list as one of the following:

- Top, which aligns icons at the top of the view. This is the default alignment.
- Left, which aligns icons at the left of the view.

Sort Sets the sort order for icons in the list as one of the following:

- None, which means no sort is applied. This is the default setting.
- Ascending, which sorts items in ascending order based on item text.
- Descending, which sorts items in descending order based on item text.

Auto Arrange Automatically keeps icons arranged in both the Icon and Small Icon views.

Single Selection Specifies that a user can select only one item at a time. By default, a user can select multiple items.

Share Image List Specifies that the list control box does not assume ownership of the image lists assigned to it; that is, the image lists are not destroyed when the list control box is destroyed. This allows the same image list to be used with multiple list-control-box view controls.

No Label Wrap Displays item text on a single line in Icon view. By default, item text may wrap in Icon view.

Edit Labels Allows item labels to be edited in place. To support this, the parent window must process the `LVN_ENDLABELEDIT` notification message.

Owner Draw Fixed Allows the owner window to paint items in the Report view. The list-control-box view control sends a `WM_DRAWITEM` message to paint each item but does not send separate messages for each subitem. The `itemData` member of the `DRAWITEMSTRUCT` structure contains the item data for the specified list-control-box view item.

No Scroll Disables scrolling; all items must appear within the client area.

No Column Header Specifies that no column header is displayed in the Report view.

No Sort Header Prevents column headers from acting like buttons. Commonly, clicking a column head sorts the list by the column entries, but clicking may be implemented for some other action. If no action is provided as a response to a column-header click, setting this option will prevent a screen response.

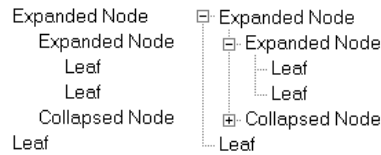
Border Creates a border around the list-control-box view control. This option is selected by default.

Tree Control Boxes

Tree control boxes are used to display hierarchical information in a tree format, where branches can be collapsed or expanded. The branches in the tree control may be displayed as a simple indented list or complete with node buttons and lines. Two styles of tree control boxes are illustrated in Figure S7.19.

FIGURE S7.19:

Two formats for tree controls

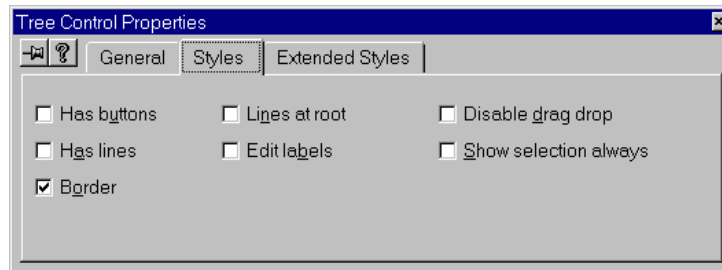


When a tree control box is created, the default ID value `IDC_TREEn` is supplied. You can enter a new ID or select one from a list of defined IDs in the General tab of the Tree Control Properties dialog box. This dialog box contains the properties listed in the earlier section titled “General Properties for Other Controls.”

In addition to the General properties for the tree control box, the Styles properties allow you to set the control’s appearance. Figure S7.20 shows this tab.

FIGURE S7.20:

Setting tree control box styles



The Styles tab contains the following properties for tree control boxes:

Has Buttons Displays plus (+) and minus (–) buttons next to parent items in the tree. These can be used to expand or collapse a parent item’s list of child items. To include buttons with items at the root of the tree view, the Lines at Root option must be selected.

Has Lines Uses lines to show the hierarchy for tree items.

Border Creates a border around the tree control box.

Lines at Root Uses lines to link items at the root of the tree control. The Lines at Root option is ignored if the Has Lines option is not selected.

Edit Labels Allows the user to edit the labels of tree control items.

Disable Drag Drop Prevents the tree control from sending TVN_BEGIN-DRAG notification messages.

Show Selection Always Uses the system highlight colors to draw the selected item.

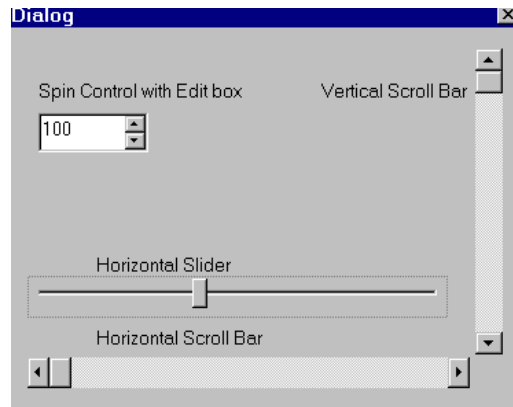
Range and Adjustment Controls

Standard dialog box features include horizontal and vertical scrollbars. Rather than being used to scroll the display within a window, dialog box scrollbars are often used as range slider controls or sometimes as range meters. For example, the Control Panel's Colors dialog box uses scrollbars to adjust the RGB intensities for custom colors. In like fashion, scrollbars might be used in a MIDI control application to set tone, voice, fade, and reverb.

More modern control versions adapted from scrollbars include slider and spin controls. In addition, the progress control, while not directly adapted from a scrollbar, shares some of the same characteristics. Figure S7.21 shows some examples of dialog box scrollbars and other range and adjustment controls.

FIGURE S7.21:

Scroll bars, sliders, and spin controls



TIP

Remember, edit boxes, list boxes, and combo boxes supply their own scrollbars. They do not require separate provisions.

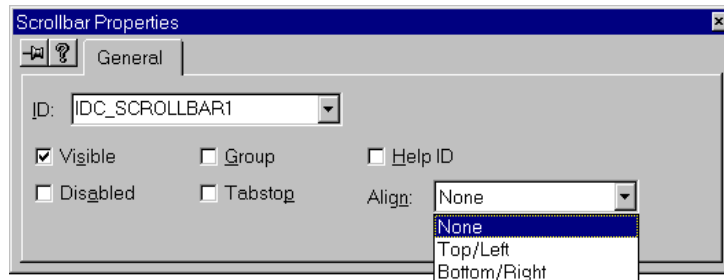
Horizontal and Vertical Scrollbars

Even though horizontal and vertical scrollbars are represented by different buttons on the dialog box editor's toolbar, these buttons are essentially a single tool differing only in their orientation. Both versions return the same event messages and respond to the same instructions (as do, incidentally, the slider and spin controls). The only real difference between the two forms of scrollbars is whether the `SB_HORZ` or `SB_VERT` argument is used when the scrollbar is generated.

When a scrollbar is created, a default ID value is supplied: `IDC_SCROLLBARn`. You can enter a new ID or select one from a list of defined IDs in the General tab of the Scrollbar Properties dialog box, as shown in Figure S7.22.

FIGURE S7.22:

The General tab of the Scrollbar Properties dialog box



This dialog box contains the properties listed in the earlier section titled "General Properties for Other Controls," along with one additional property, **Align**. The options for the **Align** property include the following:

None No special sizing or alignment is performed. The size of the scrollbar is the size specified in the resource script. This alignment is the default.

Top/Left The scrollbar is set to a standard width (thickness) and aligned with the upper-left corner of the scrollbar window specified in the resource script. The scrollbar length is not changed.

Bottom/Right The scrollbar is set to a standard width (thickness) and aligned with the lower-right corner of the scrollbar window specified in the resource script. The scrollbar length is not changed.

NOTE

Selecting either Top/Left or Bottom/Right alignment sets the thickness of the scrollbar to match the width of the scrollbar's endpads and thumbpad.

Sliders

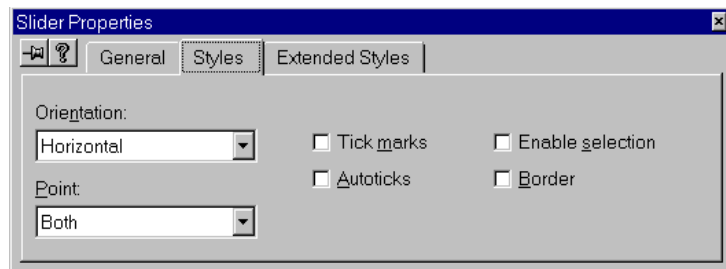
A slider control (also called a *trackbar*) is a scrollbar where the endpads of the conventional scrollbar are lost and the thumbpad is replaced by a choice of slider tabs. Slider controls may also include tick marks along their length, have slider tabs pointed to one side or the other, and be vertically or horizontally oriented.

The default resource ID for a new slider control is `IDC_SLIDERn`. You can change this ID and set the other general properties for the slider control through the Slider Properties dialog box. The General tab of this dialog box lists the same properties described earlier in “General Properties for Other Controls.”

The Styles tab of the Slider Properties dialog box allows you to set orientation, tick marks, and other options. This tab is shown in Figure S7.23.

FIGURE S7.23:

Setting slider control styles



The Styles tab contains the following properties for slider controls:

Orientation Displays the slider (trackbar) with a Horizontal (default) or Vertical orientation.

Point Displays tick marks (if the Tick Marks property is enabled) on either or both sides of the slider and alters the slider knob. The tick marks and knob have the following orientations:

- Both, which displays tick marks on both sides of the slider. The slider knob is rectangular. This is the default setting.
- Top/Left, which displays tick marks on the top of a horizontal slider or on the left of a vertical slider. The slider knob changes to point to the selected side.
- Bottom/Right, which displays tick marks on the bottom of a horizontal slider or on the right of a vertical slider. The slider knob changes to point to the selected side.

Tick Marks Enables the display of tick marks on a slider.

Autoticks Sets a tick mark at each increment in the slider's range of values. Tick marks are created automatically by sending the `TBM_SETRANGE` message.

Enable Selection Changes the narrow slider (seen earlier in Figure S7.21) to an open bar that can display a selection range with triangles and a highlighted area.

NOTE

Selection limits are not created during dialog box design but may be set during execution. This means that the triangles and highlight area will not appear during testing with the dialog box editor.

Border Creates a border around the slider control.

Spin Controls

A spin control is a second variation of the scrollbar. In this adaptation, only the endpads remain in the form of two buttons; the body of the scrollbar and the thumbpad have vanished. The default form of a spin control is vertically oriented, with up and down arrows on the buttons. The alternative is a horizontal control, with the arrow buttons pointing right and left.

The spin control has a set range (established by the application). It may wrap values when the limits are reached or simply stop when the limits are encountered.

While you can use a spin control by itself, the spin button is commonly linked to an edit box (buddy window), as illustrated earlier in Figure S7.21, or to a static text field. In these cases, the spin control operations are automatically reflected in the buddy window.

NOTE

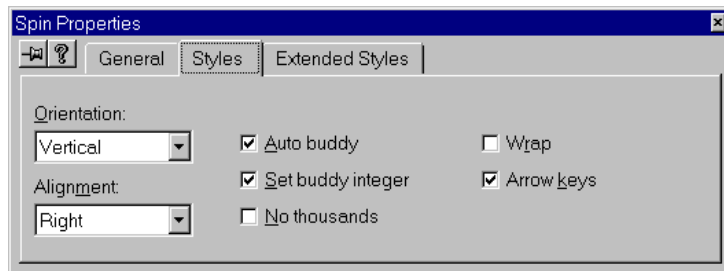
When an edit box is used as the buddy window and the `CSpinButton` class is used, changes in the edit box value are automatically reflected in the spin control's value.

Spin controls have the same General tab properties as the other types of controls (see the section “General Properties for Other Controls,” earlier in the chapter). The default resource ID for a new spin button control is `IDC_SPINn`.

On the Styles tab of the Spin Properties dialog box, you can specify the control's orientation, alignment, and other options. Figure S7.24 shows this tab.

FIGURE S7.24:

Setting spin button styles



The Styles tab contains the following properties for spin controls:

Orientation Sets the spin control display as Vertical (up/down, the default) or Horizontal (right/left).

Alignment Sets the position where the spin control appears (on execution) relative to the buddy window. If no buddy window is established, alignment is irrelevant. The alignment value can be one of the following:

- Unattached, so that the spin control is not associated with any other control. This is the default setting.
- Left, so that on execution, the spin control is positioned next to the left edge of the buddy window. At the same time, the buddy window is moved right and the width adjusted to accommodate the width of the spin control.

- Right, so that on execution, the spin control is positioned next to the right edge of the buddy window. At the same time, the buddy window is moved left and the width adjusted to accommodate the width of the spin control.

TIP

Selecting Left or Right alignment does not change the layout of the controls during resource editing. The selected alignment appears only during execution.

Auto Buddy Selects the previous window in the Z order (tab order) automatically as the spin control's buddy window. In turn, the buddy window displays the values (as text) set by the spin control. Normally, the buddy window will be an edit box or a static text field.

Set Buddy Integer Sets the text of the buddy window using the `WM_SETTEXT` message when the up or down buttons of the spin control are clicked. The text may have the setting value formatted as a decimal or hexadecimal string.

No Thousands Prevents the buddy window from inserting a thousands separator between every three digits in decimal format.

Wrap Wraps the spin-control value when it is incremented or decremented beyond the ending or beginning of the range.

Arrow Keys Increments or decrements the value of the spin control automatically when the up or down arrow keys are pressed. This option is checked by default.

NOTE

In theory, a spin control could be "buddied" with any other type of control. For example, the label on a button might display the spin-control value, or a spin control might serve as a fine control for a slider or scrollbar. In practice, however, you will probably need to write your own provisions to make a spin control interact appropriately with anything except a numeric edit box or static text field.

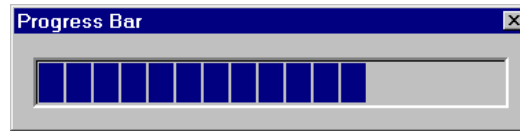
Progress Bars

The progress bar is not a scrollbar derivative per se, even though it does have a similar appearance and, in the past, scrollbars were sometimes used to provide

progress bar displays. The progress bar control is not a control in the usual sense. It does not react to user input, nor send messages to the application; its function is to show the progress of a task. For example, progress bars are commonly used when installing new software products. Figure S7.25 shows an example of a progress bar.

FIGURE S7.25:

A progress bar



The general properties for a progress bar are the same as those on the General tab of the other controls' Properties dialog boxes (see "General Properties of Other Controls," earlier in the chapter). There is one addition, however: the Border property, which creates a border around the trackbar control. The Border option is checked by default.

Other Control Types

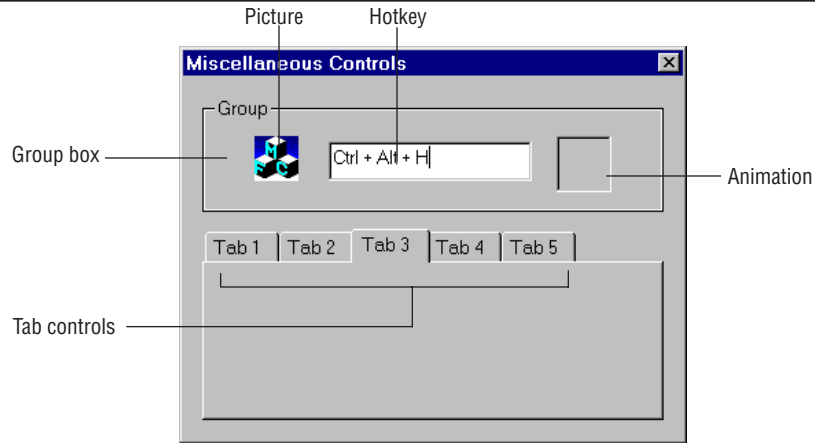
The remaining control features are used to customize the appearance of a dialog box. You can add icons or bitmaps, provide visual grouping, and use shading to enhance the appearance of the dialog box. These controls include the following:

- Group boxes
- Hotkey controls
- Tab controls
- Pictures (bitmaps and icons)
- Animated controls
- Custom controls

With the exception of custom controls, each of these is illustrated in Figure S7.26.

FIGURE S7.26:

Other control features



Group Boxes

Group boxes are simply outline boxes used to visually group controls by enclosing one or more controls in an outline with an optional group title.

The general properties for a group box are the same as for the other types of controls (see “General Properties for Other Controls,” earlier in the chapter). However, group boxes also have a Caption property, which provides a label that appears in the upper-left corner of the group box frame.

The default resource ID for a group box is IDC_STATIC. This is the same ID used for a static text field—and for the same reason: A group box normally is not selectable and is not expected to receive or return messages.

Through the Styles tab of the Group Box Properties dialog box, you can set the horizontal alignment, add an icon or a bitmap, and set other properties. Figure S7.27 shows this tab.

FIGURE S7.27:

Setting group box styles



The Styles tab contains the following properties for group boxes:

Horizontal Alignment Sets the position of the group box's caption text to the Center, Right, or Default (left) position.

Icon Indicates that the group box title displays an icon. The Caption field in the General tab identifies the icon to display.

Bitmap Indicates that the group box title displays a bitmap. The Caption field in the General tab identifies the bitmap to display.

Notify Notifies the parent window when the user clicks or double-clicks a group box.

Flat Gives the group box a flat appearance, without three-dimensional shading.

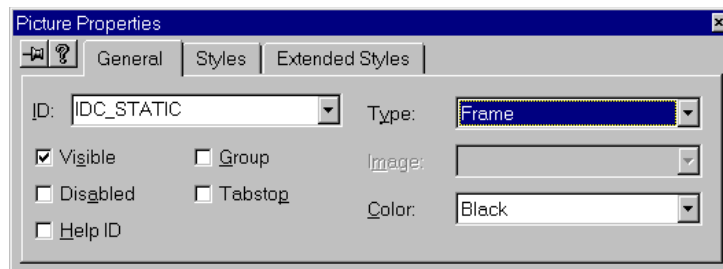
Pictures

The picture control is a static control element that does not respond to mouse selection and, by default, does not return any event messages. Instead, picture controls are customarily used simply to insert a graphic of some form into a dialog box.

Figure S7.28 shows the General tab of the Picture Properties dialog box. This tab has the same general properties as the other types of controls (see “General Properties for Other Controls,” earlier in this chapter), plus a few extra properties.

FIGURE S7.28:

The General tab of the Pictures Properties dialog box



The following general properties are specific to pictures:

ID Sets the ID for a picture control. The default resource ID for a picture control is `IDC_STATICn`, the same as for a static text field or group box. If you want the picture to function as an active control—for example, as a

button—the control type should be icon, bitmap, or metafile, and you should replace the default ID with a unique identifier. The new resource ID may be a symbol, an integer, or a quoted string.

Type Sets the type of static graphic to display as one of the following:

- Frame, which displays a empty rectangle in the color specified in the Color property. Like a group box, a frame may be used to visually group controls. This is the default type.
- Rectangle, which displays a filled rectangle in the color specified in the Color property.
- Icon, which displays an icon in the dialog box. The identifier of the icon is specified in the Image property.
- Bitmap, which displays a bitmap in the dialog box. The identifier of the bitmap is specified in the Image property.
- Enhanced Metafile, which displays an enhanced metafile in the dialog box. The application must provide the means of identifying and executing the metafile.

NOTE

A *metafile* is a graphic image recorded as a series of instructions for its creation. Metafiles are discussed in Supplement 16.

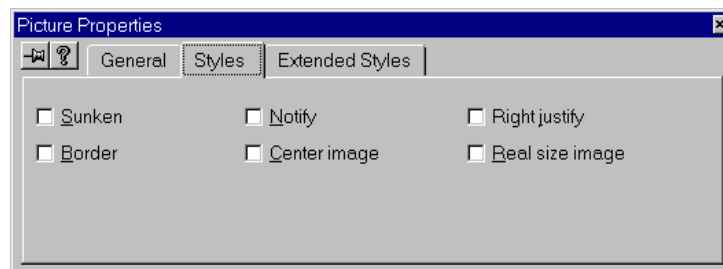
Image Provides the identifier for the icon or bitmap to display.

Color Sets the color of a frame or rectangle to Black (the default), White, Gray, or Etched (which provides a three-dimensional appearance).

Figure S7.29 shows the Styles tab of the Picture Properties dialog box, which has properties for controlling the appearance of the picture.

FIGURE S7.29:

Setting picture styles



The Styles tab contains the following properties for pictures:

- Sunken** Creates a border with a sunken edge around the picture control.
- Border** Creates a border around the picture. This option is selected by default.
- Notify** Allows the picture to notify the parent window when it's clicked or double-clicked.

TIP

Setting the Notify property allows an icon or bitmap image to respond as if it were a button. When setting the Notify property, however, you should change the default ID to a unique identifier to ensure proper handling and recognition of the message.

- Center Image** Fills the rest of the client area with the color of the pixel in the top-left corner of the bitmap or icon if the bitmap or icon is smaller than the client area of the picture control.
- Right Justify** Sets the lower-right corner of a picture control to remain fixed when the control is resized; only the top and left sides are adjusted to accommodate.
- Real Size Image** Specifies that a static icon or bitmap control will not be resized as it is loaded or drawn. If the image is larger than the destination area, the image is clipped.

Hotkeys

A hotkey control is a simple input box in which a user can select a hotkey combination to assign to a particular task. When the hotkey control is selected, any key combination pressed is displayed as a hotkey combination. Figure S7.26, shown earlier, illustrates an example where pressing the *H* key while holding down the Ctrl and Alt keys is recognized as Ctrl+Alt+H.

The Ctrl, Alt, and Shift keys may be combined with any of the following:

Alphanumeric keys	Insert	Right	PgUp
Function keys	Delete	Left	PgDn
Number pad keys	Home	Up	CapsLock
Scroll Lock	End	Down	NumLock

Keys or combinations that cannot be used include:

Ctrl+Alt+Delete	Escape	Backspace	Tab
Windows key	System key	Enter	/ (on number pad)

WARNING

The hotkey control does apply minimum validation and will refuse some key combinations when entered. However, this control does not provide complete validation and, except for certain system hotkey codes, does not check the hotkey assignments against other conflicting assignments. Be sure to avoid assigning the same hotkey twice in a dialog box.

The default resource ID for a group box is `IDC_HOTKEYn`. The general properties for a hotkey control are the same as for the other controls (see “General Properties for Other Controls,” earlier in the chapter), with the addition of a **Border** property. When the **Border** property is checked (the default), an outline appears around the hotkey control.

Animation Controls

An animation control is used to play back an animated sequence. This could be a simple sequence of images or a more elaborate video sequence. (But keep in mind that AVI sequences require a lot of storage space.)

The default resource ID for a group box is `IDC_ANIMATEn`. The general properties for an animation control are the same as for other controls, with these additions:

Center Centers the animation in the animation control window.

Transparent Draws the animation using a transparent background rather than the background color specified in the animation clip.

Auto-play Sets the animation to begin playing as soon as the animation clip is opened.

Border Creates an outline around the animation control. This option is checked by default.

Tab Controls

A tab control (see Figure S7.26, shown earlier) offers a method of arranging groups of controls in a dialog box as a sequence of tabs, with each tab containing a separate set of controls. The tab control may fill the dialog box or may use only a part of the space, leaving the remainder of the space for controls common to all tabs.

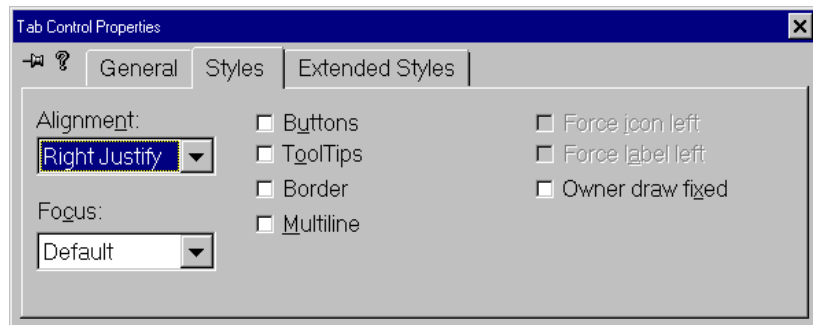
The number of tabs in a tab control and the labels for the tabs are set by the application, and a dialog box can contain several rows of tabs. The contents of the tabs are created as separate dialog boxes without title bars, system menus, and so on, and are then assigned to tabs by the application.

The general properties for a tab control are the same as for the other controls. The default resource ID for a tab control is `IDC_TABn`.

On the Styles tab of the Tab Control Properties dialog box, shown in Figure S7.30, you can set the alignment, focus, and other features of the tab control.

FIGURE S7.30:

Setting tab control styles



The Styles tab contains the following properties for tab controls:

Alignment Sets the tab control alignment as one of the following:

- **Right Justify**, which adjusts the width of each tab so each row of tabs fills the entire width of the tab control. This is the default alignment.
- **Fixed Width**, which sizes all tabs to the width of the widest label.
- **Ragged Right**, which is used with multiline tabs; tab widths are not adjusted to fill the rows.

Focus Determines how tabs are selected; choose from one of the following:

- Default, so keyboard selection may be used to select a tab or a tab may be selected by clicking with the mouse. (And of course, Default is the default setting.)
- On Button Down, so tabs receive the input focus (come to the front) when the tab is clicked.
- Never, so tabs do not receive the input focus when clicked. Instead, tabs must be selected by the application.

Buttons Makes the tabs in the control resemble buttons. When tabs are displayed in button format, they should perform the same function as button controls; clicking a tab should carry out a task rather than display a tab page.

ToolTips Creates a tooltip for each tab in the tab control.

Multi-line Displays multiple rows of tabs.

Border Creates an outline around the tab control.

Force Icon Left Left-aligns the icon; the label remains centered.

Force Label Left Left-aligns both the icon and the label.

WARNING

The reference to icons in the Force Icon Left and Force Label Left options is something of a mystery since selecting either option expands the tabs—a feature that is not mentioned—and the icons referred to are unknown. You can experiment for yourself to see how these properties work.

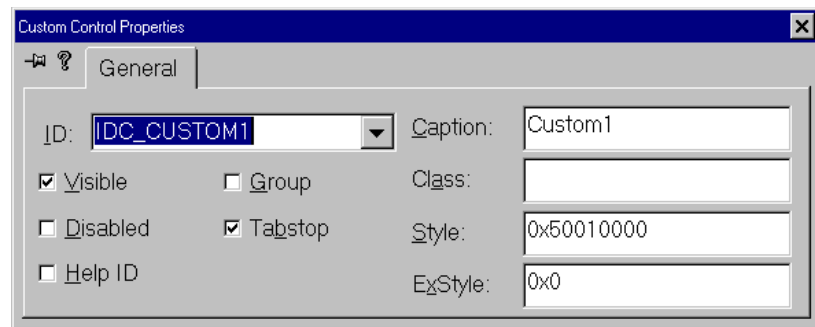
Owner Draw Fixed Makes the parent window responsible for drawing tabs in the control.

Custom Controls

The custom control is undefined but provides a placeholder—a dark gray rectangle—representing a custom-control element. It is the responsibility of the application (or a custom library) to handle the appropriate screen display, to issue and respond to messages, and to handle any other required interactions. The General tab of the Custom Control Properties dialog box is shown in Figure S7.31.

FIGURE S7.31:

The Custom Control Properties dialog box



The default resource ID for a custom control is `IDC_CUSTOMn`. The general properties for a custom control are the same as for the other types of controls, with the following additional properties:

Caption Sets a string entry that appears as a label for the custom control. To make one of the letters in the caption a mnemonic key (hotkey), place an ampersand (&) directly before the letter. A default caption name is supplied in the format "Caption *n*" (*n* represents a number matching the resource identifier).

Class Sets the name of the control's Windows class. The named class must be registered before the dialog box containing the control is created. (During execution, the class must be registered before calling the dialog box.)

Style Sets a 32-bit hexadecimal value specifying the control's style. This is primarily used to edit the lower 16 bits making up a user control's substyle.

ExStyle Sets a 32-bit hexadecimal value specifying the control's extended style.

NOTE

The Style and ExStyle properties are undefined until a custom control is created. How or if these properties are used depends on the design and functionality of the custom control.

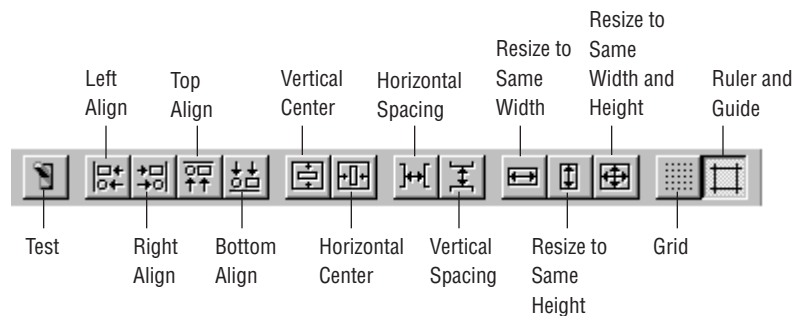
Alignment, Positioning, and Sizing Tools

You can position and size dialog box controls by using the mouse or, in some cases, by entering position and size information directly, but the easiest way to align controls is to use the tools provided by the resource editors.

Most of the Microsoft Developer Studio tools are accessible from its toolbar. Figure S7.32 shows this toolbar, with labels to identify the buttons.

FIGURE S7.32:

The Microsoft Developer Studio toolbar



The toolbar buttons work as follows:

Test button Allows you to test a dialog box during development, before the dialog box becomes part of an application.

Alignment buttons Provide left, right, top, and bottom alignment. To align a group of controls, hold down the Shift key while clicking each control. The last control selected will become the reference control, and all other selected controls will be aligned to the position of that control.

Centering buttons Provide vertical and horizontal centering. These can be used to center one or more controls relative to the dialog box itself. If you select more than one control, the selected controls are centered as a group but retain their positions relative to each other.

Spacing buttons Provide horizontal or vertical spacing. These buttons give you the ability to select two (or more) controls and then space them equally across or down. If you are spacing controls horizontally, the leftmost and rightmost extremes are taken as the limits, and all controls selected are spaced equally between these two limits, without affecting

individual vertical positioning. For vertical spacing, the highest and lowest extremes are taken as limits, and horizontal spacing remains unaffected.

Sizing buttons Allow you to resize controls. Again, you must select more than one control, and the last control you select (identified by the dark handles on the outline) serves as the size reference. Multiple controls may be resized to the same width, to the same height, or with both height and width adjusted at the same time.

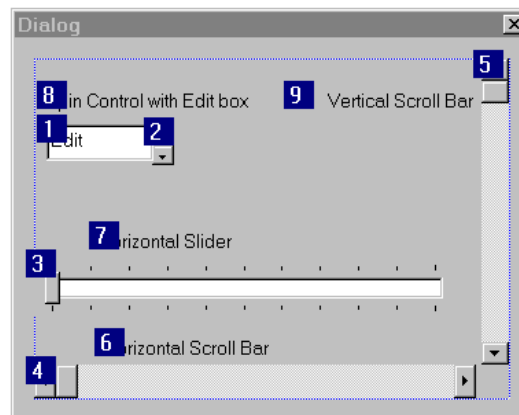
Grid button Provides a background grid of fine dots, which can help you align controls. When the grid is enabled, the ruler and guides are disabled. Then, when you move a control, its position is snapped to the grid. Likewise, when you resize a control, the size is snapped to the grid.

Ruler and Guide button Provides vertical and horizontal rulers in dialog box units, plus a guide that appears as a faint blue border or margin around the dialog box. When the guide is enabled, controls moved toward the guides tend to snap to the guide if they are close enough. From the Layout menu, select the Guide Settings option to change the guide settings and adjust grid spacing.

To adjust the tab ordering for dialog box controls, select Tab Order from the Developer Studio's Layout menu. When you select this option, all of the dialog box controls are labeled with a tab order number, as shown in Figure S7.33. To change the tab order, click the first control and then continue clicking controls in the order desired. The tab order numbers will change to reflect the new order. To stop setting the tab order, simply click anywhere except on a control.

FIGURE S7.33:

Selecting the Tab Order option from the Developer Studio's Layout menu shows your dialog box with tab order numbers.



Dialog Box Testing Tips

When testing a dialog box during development, a few items to check in particular include the following:

- **Tab order for controls.** Use the Tab key to move between controls and make sure that the required groups, tab stops, and ordering are appropriate.
- **Radio button groups.** Be sure that radio button groups function correctly and that buttons reset appropriately. If there is a problem, check the tab order.
- **Overlapping elements.** Check particularly for static text elements that may be larger than the text they contain and may overlap (and conceal) other elements.

Three Dialog Boxes for FileView1



The *FileView1* demo requires three dialog boxes: About, File Type, and Open File. These three dialog boxes are described in the following sections and are included in the *FileView1.RC* resource script on the CD accompanying this book. However, even though you do not need to re-create these dialog boxes using a dialog box editor, they illustrate some important features of dialog box construction.

NOTE

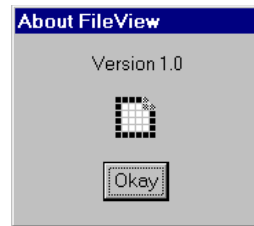
The *FileView2* demo uses the common dialog File Open facility, so it includes only one dialog box resource: the About dialog box. The script for this resource is included on the CD accompanying this book and needs no explanation.

The About Dialog Box

The About dialog box, shown in Figure S7.34, consists of a captioned dialog box without a system menu. The title bar bears the text “About FileView.” Three dialog box elements appear as a centered text line, an icon box, and a single button.

FIGURE S7.34:

A simple About dialog box
with an icon



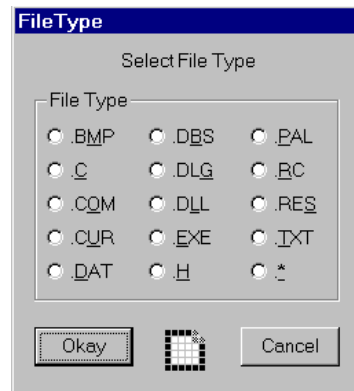
The single button returns a value, `IDOK (0x01)`, which is defined in `Windows.H`.

The File Type Dialog Box

The File Type dialog box, shown in Figure S7.35, again uses a captioned dialog box with a static text instruction supplementing the caption. In this example, two control buttons appear at the bottom on either side of the File Type icon. The Okay button, as the heavy outline illustrates, is the default.

FIGURE S7.35:

The File Type dialog box



Inside the group box, 15 auto radio buttons offer a choice of file extensions. When the dialog box is initialized by the *FileView* demo, the `.*` button will be set as the default extension, but this is a provision of the program code, not the dialog box.

Also, even though the auto radio buttons are enclosed by a group box outline, this is for visual purposes only; the group box here does not control the grouping. Instead, the grouping is assigned by setting the Group property for the first item in the group and setting the tab order for all of the buttons.

NOTE

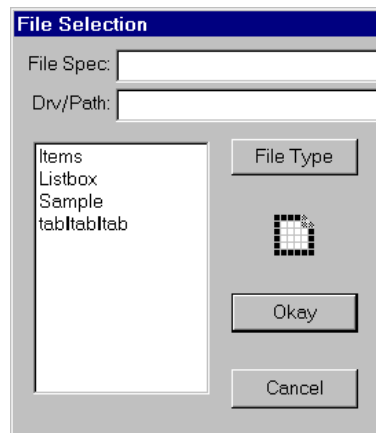
The resources and dialog boxes created for the *FileView1* demo may be edited with either the Microsoft or Borland resource editor. The *FileView2* application files, however, are not compatible with the Borland compiler or resource editors.

The File Selection Dialog Box

In some respects, the File Selection dialog box used by the *FileView* demo is the most complicated and the most important. This dialog box, shown in Figure S7.36, offers two edit boxes, plus a list box where files matching the file specification will be displayed (together, of course, with directory and drive IDs). In addition to the customary Okay and Cancel buttons, a third button, File Type, is provided to call the File Type dialog box.

FIGURE S7.36:

The File Selection dialog box



Beginning at the top, the File Spec edit box has been provided with a default test string, `"*.*"`. This text string, however, is superfluous; when the dialog box is initialized, the application will provide its own string both here and in the Drv/Path edit box below the File Spec edit box.

Next, when the dialog box is initialized, the list box will be filled with file, directory, and drive information using the application-assigned, default file specification and the current (active) drive/path settings.

Last, the File Type button returns an `IDM_TYPE` message, which is used to instruct the application to load the File Type dialog box.

This completes our discussion of working with dialog box resources. We've covered all types of dialog box controls. Custom dialog box controls have been mentioned only briefly, as alternatives to the standard controls. Fortunately, the majority of applications will not require custom control designs.

Also, as you may have noticed, no mention has been made of adding menus to dialog boxes. This is not because menus are not supported, but because the dialog box editor is not the appropriate mechanism for constructing menus. We'll cover menu editors and menu construction in the next chapter.

S U P P L E M E N T

E I G H T

S8

Menu Resources

- Menu editor features
- Menu item properties
- Menus in dialog boxes
- Menu scripts

For many Windows applications, the primary entry point is a menu bar that offers initial options and access to principal features. In other cases, dialog boxes may use menus to present further choices or applications may present different primary menus depending on the current operation.

In this chapter, we'll look at how menus are constructed and some of the possible arrangements for menus and submenus.

A Menu Editor

Menu resources are easy to create, and, in keeping with their text nature, require little more than ASCII scripts for their design. However, menu editors provide an easier way to construct and test menus. Menu editors can generate either an .RC menu script or a compiled .RES resource.

In the Microsoft Developer Studio menu editor, the menu is presented in a single window, and menu item properties are displayed in the pop-up Properties dialog box. Figure S8.1 shows a menu under construction in this menu editor.

Here you should notice two blank rectangles: one at the end of the primary menu bar and one at the bottom of the pull-down File menu. Neither of these items actually appears in the menu; they are simply placeholders, ready for your new entries.

To add a menu item, simply double-click the blank rectangle on the primary menu bar or on the pull-down menu. This brings up the Menu Item Properties dialog box for the item, as shown in Figure S8.2.

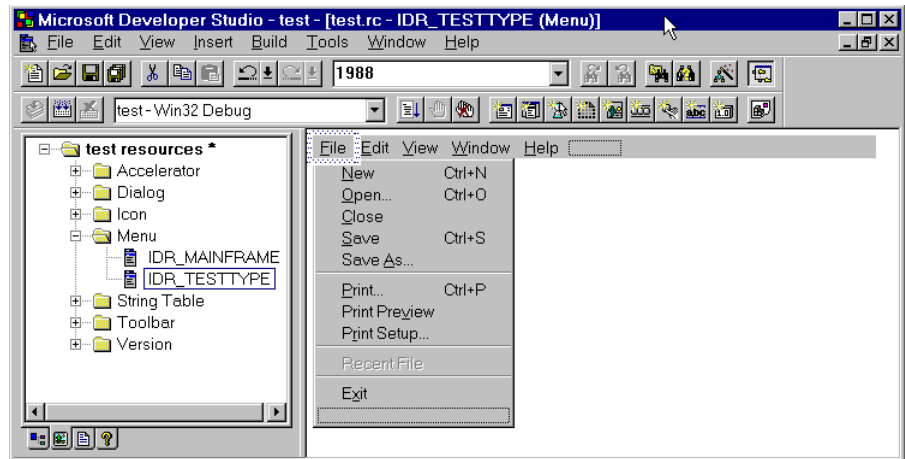
In the Menu Item Properties dialog box, enter an ID and caption (if appropriate) or select an option for the entry. A new blank item will appear as soon as you have entered the new caption or selected the Separator option. See the "Menu Item Properties" section later in the chapter for details on setting properties for your menu items.

To move a menu item to a different position in the menu, simply click and drag the entry to the position desired.

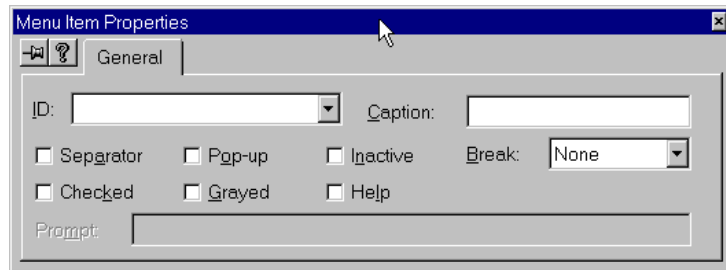
To remove a menu item, highlight the item and press the Delete key.

FIGURE S8.1:

The Microsoft Developer Studio menu editor

**FIGURE S8.2:**

The Developer Studio's Menu Item Properties dialog box



Menu Size Limitations

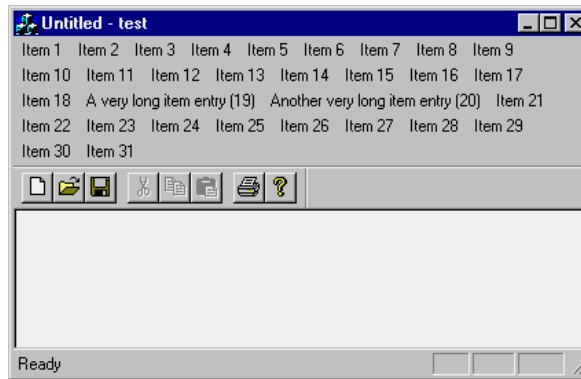
Theoretically, you can create a menu of virtually any size. But in practical terms, too large a menu probably means that you need to rethink your menu's organization.

As an unwieldy example, Figure S8.3 shows a primary menu bar with a total of 31 entries, requiring five lines to display on a screen 420 pixels wide.

Obviously, there must be some actual limits on the number of menu elements and the number of levels supported, although these limits have nothing to do with the resource compiler or Windows 2000. To set practical limits, consider how much is too much. At what point does a menu bar become too complex to be comfortable to use? When does it become a liability instead of an asset?

FIGURE S8.3:

An unwieldy primary menu

**NOTE**

If you reduce the width of the window, the menu bar will be rearranged automatically to use as many lines as necessary. (This automatic rearrangement does not happen for toolbars, however.)

These questions apply not only to the number of primary entries but also to the number of levels and pop-up (pull-down) submenus.

As a general rule of thumb, the primary menu should not exceed one line (in a normal-size window), and the number of pop-up submenu levels should probably not go further than two or three. If your application needs more items than a simple menu structure can supply, consider creating a very simple primary menu, with the menu selections calling a series of dialog boxes to offer the more complex options.

TIP

For an alternative to overloaded menus, consider using pop-up menus. See Chapter 5 in the book for details.

Text for Menu Entries

Individual menu entries can be as simple as a single word, a brief phrase, or some combination of these. They may or may not include hotkey assignments or accelerator keys.

The text for a menu entry is one of its properties (the Caption property). In the Microsoft Developer Studio menu editor, enter the text in the Caption field of the Menu Item Properties dialog box. (You will read more about setting menu properties in the “Menu Item Properties” section later in the chapter.)

Note that you cannot enter a tab character directly by pressing the Tab key while typing in the menu item text. The C convention `\t` is recognized as a tab instruction and is commonly used to identify accelerator keys by setting them flush-right in the menu entry. (Accelerator keys are covered briefly in the next section and in detail in Supplement 9.) The `\a` instruction also causes the text following it to be right-justified.

Also note that no provisions are made for multiline menu entries, and carriage returns and line-feeds are not supported. Thus, even though the menu editor can accept entries up to 255 characters, this does not mean that the menu can display strings of that length (but certainly you should feel free to experiment).

Menu Hotkeys

The primary menu hotkeys, which are identified in the menus by the underscore character and in the script by the ampersand character (&), are automatic assignments generated when the menu script is compiled. By custom, most menu entries use the first letter in each entry as the hotkey; however, the hotkey does not need to be the first character. Also, Windows does not underscore the entry unless it is specifically instructed to do so.

For example, to select the *O* in *Open...* as the hotkey for that menu entry, type the text as `&Open....` This entry will then appear in the menu as `O`pen....

Within any menu or submenu, you cannot use the same hotkey twice at the same level. Thus `Cu`t, `C`opy, and `C`lose Clipboard—items that are all on the same level—use three different hotkeys (*T*, *C*, and *L*). However, if you added a `S`earch entry that called a pop-up menu that had a `S`top entry, the *S* hotkey could be used again because these items are on different levels.

TIP

If more than one ampersand appears in a menu item, only the last ampersand entry is recognized. If you want to include an actual ampersand in the menu entry text, enter a double ampersand (`&&`). For example, to create the menu item `S`earch & `R`eplace, enter `S`earch `&&` `R`eplace. The ampersand will appear in the item as a single & character and will not be treated as a hotkey identifier.

In many cases, menu entries have secondary hotkeys identified as *Shift+key* or *Ctrl+key*. Another convention uses a simple syntax to identify hotkeys, with the caret character (^) indicating the Ctrl key. Both conventions are acceptable; however, simply identifying a secondary hotkey in the menu definition does not assign the key definition as a functional hotkey. These hotkey assignments are handled as accelerator keys.

Accelerator hotkeys, which are global, cannot be duplicated within a dialog box or application window, although the same accelerator key combination can be used for different purposes in different menus or in different dialog boxes. The use of accelerators is covered in the next chapter.

Menu Item Properties

To set menu item properties in the Microsoft Developer Studio menu editor, double-click a menu item to bring up the Menu Item Properties dialog box (see Figure S8.2, earlier in the chapter).

The following menu item properties generally apply to menu items created with menu editors, although they may be labeled with different names (for example, the identifier property is named *ID* in the Microsoft Developer Studio menu editor and *Name* in the Borland C++ Builder menu editor).

ID Acts as an identifier; it is commonly a mnemonic symbol, defined in the header file. Unlike other resources, new menu resources do not get a default ID. However, the prefix *IDM_* is commonly used to identify menu item messages. Pop-up menu items, which are handled internally, do not have ID values and do not return messages when selected. Similarly, menu separators, which cannot be selected, do not require ID values.

Caption Sets the menu item text (see the “Text for Menu Entries” section earlier in the chapter).

Separator Specifies that the menu item is a separator. Separator items do not have captions or IDs and cannot be selected.

Checked Specifies that the menu item is initially checked when the menu opens.

Grayed Sets the menu item as initially inactive and grayed and also sets the *Inactive* property. Before you can select an inactive (grayed) menu item, it must be enabled by the application (see *CMenu::EnableMenuItem*).

Prompt Supplies text to appear in the status bar when this menu item is selected. The prompt entry is added to the resource string table using the same ID as the menu item.

NOTE

Unlike toolbar buttons, menu items do not support tooltips. Do not add a tip entry to the prompt string entry.

Pop-up Sets the menu item as a pop-up item; that is, the primary item for a pop-up submenu. This is the default setting for top-level menu items.

Inactive Sets the menu item as initially inactive, but *not* grayed. Before you can select an inactive menu item, it must be enabled by the application (see `CMenu::EnableMenuItem`).

Help Right-justifies the item on the menu bar at runtime, but not during editing.

Break Sets the break style as one of the following:

- None, for no break (the default).
- Column, for a static menu-bar item that you want to place on a new line. For a pop-up menu, the item is placed in a new column with no dividing line between columns. Setting the Column property affects the menu only at runtime, not during editing.
- Bar, for a static menu-bar item that you want to place on a new line. For a pop-up menu, the item is placed in a new column with a vertical dividing line between columns. Setting the Bar property affects the menu only at runtime, not during editing.

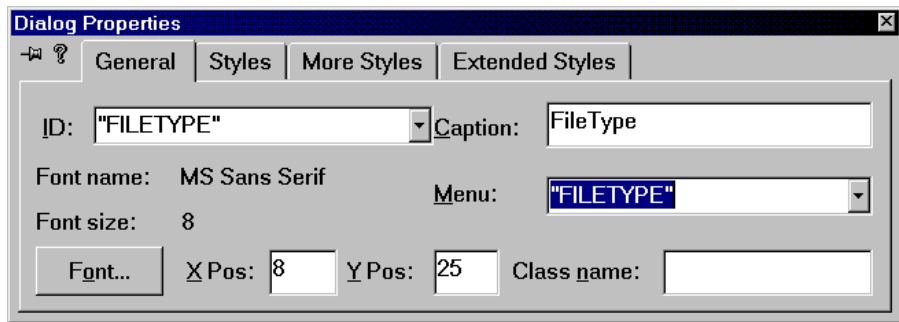
Adding Menus to Dialog Boxes

Menu resources can also be attached to dialog boxes quite easily. To add a menu to a dialog box, first create the menu as a resource.

Then, open the Dialog Properties dialog box for the dialog box resource and select the menu to attach from the Menu pull-down list. In Figure S8.4, the menu resource is identified by name ("FILETYPE"), but menu resources may also use mnemonic identifiers (such as `ID_FILETYPE_MENU`).

FIGURE S8.4:

Using the Dialog Properties dialog box to attach a menu to a dialog box



For an example of a menu in a dialog box, see the File Type dialog box in the *File-View1* demo (included on the CD accompanying this book, in the Supplement 10 folder).

Menu Scripts

A menu editor is a convenient tool, but it is not the only way to create menus. As an alternative, you can use any plain text editor, such as the Windows Notepad, to create a menu script. As an example, let's see how to create a script for a simple menu with four primary entries and an assortment of pull-down menus and further submenus.

The menu script begins with a name, `IDM_MENU1`, followed by the resource type identifier, `MENU`. On the next line, the keyword `BEGIN` identifies the start of the menu definition.

```
IDM_MENU1 MENU
BEGIN
```

TIP

You may see opening and closing brackets rather than `BEGIN` and `END` statements in scripts. Both formats are correct, although the Borland and Microsoft compilers may each complain about the other's syntax when moving a resource file from one compiler to the other.

The first menu item is identified by the keyword `POPUP` and followed by the text for this item. Because the pop-up entry will then be followed by at least one subentry, another `BEGIN/END` block is initiated on the next line.

```
POPUP "&Edit"
BEGIN
```

The next few lines define menu entries for the Edit submenu, each beginning with the keyword `MENUITEM`, followed by the entry text, the entry ID, and, optionally, one or more flag arguments controlling how the menu entry is initially displayed.

```
MENUITEM "C&ut\t^U",    201, CHECKED
MENUITEM "&Copy\t^C",    202
MENUITEM "&Paste\t^P",    203, INACTIVE, MENUBARBREAK
```

In this fragment, the first item on this submenu is presented with a checkmark (CHECKED). The third item uses the `INACTIVE` keyword to make the entry unselectable and the `MENUBARBREAK` keyword to cause a column break with a vertical separator bar.

Next, a new submenu entry is defined as an entry in the Edit submenu. The Search submenu has two entries, without any special features, but it does require a `BEGIN/END` pair to set off the submenu block.

```
POPUP "&Search\t^S"
BEGIN
    MENUITEM "&Find\t^F",    204
    MENUITEM "&Rep lace\t^R", 205
END
```

Following the Search submenu, a final entry is made in the Edit submenu but, this time, is disabled using the `GRAYED` keyword. The `MENUBREAK` keyword produces a column break but does not produce a vertical separator. The Clear clipboard menu entry is followed by an `END` statement to close the Edit submenu.

```
MENUITEM "C&lear clipboard\t^L", 206, GRAYED, MENUBREAK
END
```

The remaining primary menu level entries are the next `MENUITEMS`. The Help menu item uses the keyword `HELP` to set this entry flush-right on the main menu bar. The script terminates with a closing `END` statement.

```
MENUITEM "&Print", 101
MENUITEM "&File", 102
MENUITEM "&Help", 103, HELP
END
```

Now that you've seen the fragments and explanations, the entire `Menu_1` script follows, showing the overall structure and indentations (these indentations are for the programmer's benefit only and have no effect on how the menu script compiles).

```
IDM_MENU1 MENU
BEGIN
  POPUP "&Edit"
  BEGIN
    MENUITEM "C&ut\t^U",    201, CHECKED
    MENUITEM "&Copy\t^C",    202
    MENUITEM "&Paste\t^P",    203, INACTIVE, MENUBARBREAK
    POPUP "&Search\t^S"
    BEGIN
      MENUITEM "&Find\t^F",    204
      MENUITEM "&Replace\t^R", 205
    END
    MENUITEM "C&llear clipboard\t^L", 206, GRAYED, MENUBREAK
  END
  MENUITEM "&Print", 101
  MENUITEM "&File", 102
  MENUITEM "&Help", 103, HELP
END
```

Two Menus for FileView



The one menu script command that does not appear in the preceding example is the horizontal menu separator, `SEPARATOR`. This command is used in the following script, which is the script for the *FileView1* demo's menu (the *FileView1* and *FileView2* demos are discussed in Supplement 10 and included on the CD accompanying this book).

```
FILEVIEW MENU DISCARDABLE
BEGIN
  POPUP "&File"
  BEGIN
    MENUITEM "&Open...\t^O", IDM_OPEN    // = 103
    MENUITEM "&Type...\t^T", IDM_TYPE    // = 104
    MENUITEM SEPARATOR
  END
END
```



```

        MENUITEM "&About",      IDM_ABOUT    // = 102
        MENUITEM "E&xit\t^X",    IDM_QUIT   // = 101
    END
END

```

The *FileView2* demo's menu script, shown here, is slightly different.

```

IDR_MAINFRAME MENU PRELOAD DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "&Open...\tCtrl+O",      ID_FILE_OPEN
        MENUITEM SEPARATOR
        MENUITEM "E&xit",                  ID_APP_EXIT
    END
    POPUP "&View"
    BEGIN
        MENUITEM "&Toolbar",                ID_VIEW_TOOLBAR
        MENUITEM "&Status Bar",            ID_VIEW_STATUS_BAR
    END
    POPUP "&Help"
    BEGIN
        MENUITEM "&About FileView2...",    ID_APP_ABOUT
    END
END

```

Notice that both the *FileView1* and *FileView2* menu scripts are written using mnemonic identifiers rather than actual values. The constants are defined in the *FileView.H* header file.

As you learned in this chapter, menu editors provide a convenient way to create, define, and test application menus. If you prefer, you can also create menus as scripts, using any plain-text editor, such as the Windows Notepad.

S U P P L E M E N T

N I N E

S9

Accelerators, Strings, Header Files, and Version Information

- Accelerators and accelerator editors
- String tables and string table editors
- Header files for resource IDs
- Version information

In addition to the more obvious resource types we've discussed so far—images, dialog boxes, and menus—you should know about two other resources: accelerator keys and string resources. Although these are less graphic and less impressive than other resources, they are just as important for the programmer.

Another type of programmer resource is the header definition file, which is also discussed in this chapter. When used properly, header files can prevent a great many errors that would otherwise be difficult to identify.

And finally, there are version resources. Even though these are very simple, they provide a convenient location to record version, copyright, and source notes within the application.

Accelerator Key Resources

Accelerator keys offer a fast shortcut—in the form of keyboard hotkey combinations—for issuing application commands. Although conventional (DOS) programs have often provided similar services, frequently employing TSR utilities to translate individual keystrokes or key combinations into command sequences, accelerator keys take a rather different form.

One of the most important differences is that, unlike DOS-based TSRs, accelerator keys do not depend on interrupt processing by an outside application. Instead, accelerator key processing is handled internally by Windows and is part and parcel of the Windows messaging system.

As explained in Supplement 2, Windows does not send keystroke information directly to the applications. Instead, all Windows applications rely on Windows to intercept the hardware keyboard events, translate these as necessary, and then forward them in the form of keyboard messages to the appropriate application. This approach allows you to add provisions for special key combinations to generate custom messages in place of key-event messages; in Windows, this is only a minor change. More important, each application can define its own accelerator key combinations and the messages to be generated by each.

Still, no matter how convenient or how smooth this translation may be, it remains the programmer's responsibility to define these accelerator hotkey combinations and to prepare this information in a form acceptable to the resource compiler for inclusion in the application's resources.

Accelerator Key Combinations

An accelerator key definition consists of two parts:

- A keyboard key or key combination
- A message value to be sent to the application when the key combination is entered

In general, single keystrokes are not used as accelerator keys simply because these key events have other purposes that take precedence. An accelerator key is commonly defined as a key combination requiring one conventional key plus one or more of the Ctrl, Alt, or Shift keys. For example, common shortcuts for Edit menu commands are Ctrl+C for Copy, Ctrl+X for Cut, and Ctrl+V for Paste. The conventional key can be defined as either an ASCII key or as a virtual key.

Virtual versus ASCII Keys

Virtually (the pun is unavoidable) all of the keys on the keyboard—whether a standard or an enhanced keyboard—can be defined as accelerator keys using the virtual key definitions provided in the WinUser.H header file.

Not all keys, however, have ASCII equivalents, and a few ASCII keys do not have virtual key equivalents, such as the exclamation point (!). Furthermore, some virtual key definitions do not correspond to anything found on the contemporary keyboard, such as the VK_ZOOM or VK_NONAME virtual keys; some refer to a non-keyboard device, such as the VK_MBUTTON virtual key. Table S9.1 lists the key codes that you don’t want to use as accelerator keys.

TABLE S9.1: Key Codes Not Recommended for Use as Accelerator Keys

Key codes*	Comments
0Ch, 5Bh..5Dh, 60h..69h	Special requirements and functions
6Ah..6Bh, 6Dh..7Bh, A0h..A5h	Enhanced keyboards only
29h..2Fh, 2Ah..2Bh, 2Fh, 6Ch, 7Ch..87h, E5h, F6h..FEh	OEM-specific keys
05..07h, 0Ah..0Bh, 0Eh..0Fh, 1Ah, 3Ah..40h, 5Eh..5Fh, 88h..8Fh, 92h..9Fh, A6h..E4h, E6h..F5h	Not assigned
15h..19h, 1Ch..1Fh	Reserved for Kanji system

NOTE

Refer to Supplement 2 for more details on key codes and virtual key identifiers.

In general, however, an ASCII key refers to any of the alphanumeric keys that produce displayable characters on the screen. These include the punctuation keys and the spacebar.

Virtual key definitions (all of which begin with the prefix `VK_`) refer principally to the function, arrow, and keypad keys. Thus, the F1 key is defined as `VK_F1`, the PgDn key as `VK_NEXT`, the down-arrow key as `VK_DOWN`, and the left Shift key on an enhanced keyboard as `VK_LSHIFT`. The standard alphanumeric keys, however, are not excluded; they are identified as `VK_A` through `VK_Z` and `VK_0` through `VK_9`.

NOTE

Keep in mind that uppercase and lowercase keys are not differentiated either as virtual keys or when used as accelerator keys employing an ASCII key definition.

Accelerator Key Scripts



Accelerator keys can be defined, in script form, using any plain-text editor, such as the Windows Notepad. Here is a sample script for the *FileView1* demo's keyboard accelerators (the *FileView* demo is discussed in the next chapter):

```
FILEVIEW ACCELERATORS
BEGIN
    "X", IDM_QUIT, ASCII, CONTROL    // = 101
    "O", IDM_OPEN, ASCII, CONTROL    // = 103
    "T", IDM_TYPE, ASCII, CONTROL    // = 104
END
```

In this example, the three accelerator keys are Ctrl+X, Ctrl+O, and Ctrl+T, returning the values `IDM_QUIT`, `IDM_OPEN`, and `IDM_TYPE`, respectively.

In the following example, the preceding accelerator keys are repeated in a different format, together with five new accelerator keys showing various Ctrl, Alt, and Shift key modifiers.

```
ACCLDEMO ACCELERATORS
BEGIN
    VK_X, IDM_QUIT, VIRTKEY, CONTROL    // = 101
    VK_O, IDM_OPEN, VIRTKEY, CONTROL    // = 103
    VK_T, IDM_TYPE, VIRTKEY, CONTROL    // = 104
```

```
VK_F1, 105,      VIRTKEY
VK_F4, 106,      VIRTKEY
VK_F6, 107,      VIRTKEY,      SHIFT
"s ", 108,      ASCII,  ALT,    SHIFT
"G ", 109,      ASCII,  CONTROL, SHIFT
END
```

NOTE

The spacing is irrelevant to the resource compiler and has been added only to make the various elements of each definition easier to read. Also notice that it does not matter whether the ASCII key definition is entered as uppercase or lowercase, and the CapsLock status does not affect recognition of the accelerator key combination.

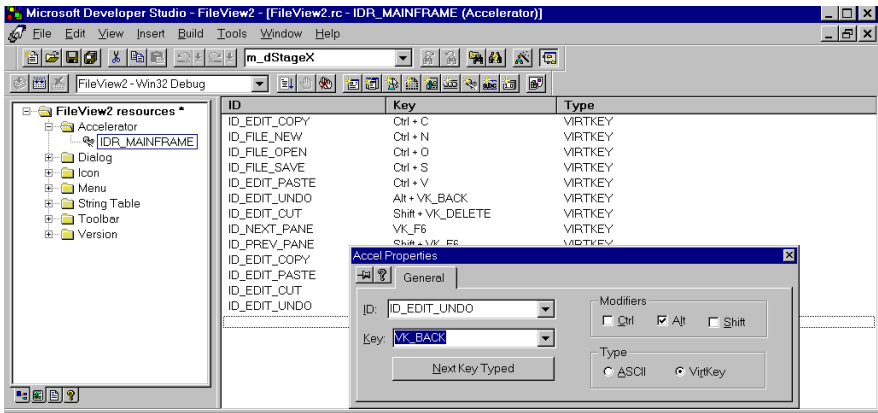
For non-ASCII keys, this virtual-key definition format is almost essential. However, a more convenient entry method is provided by an accelerator editor.

An Accelerator Editor

The Microsoft Developer Studio offers an accelerator editor, which includes the Accel Properties dialog box for entering or editing an accelerator table item. Figure S9.1 shows the accelerator editor and the Accel Properties dialog box.

FIGURE S9.1:

The Microsoft Developer Studio accelerator editor



The editor offers the following properties for accelerators:

ID Sets the resource ID, which is normally a mnemonic symbol defined in the header file, but it may also be an integer value or a quoted string.

Modifiers Indicate whether the accelerator key is a combination formed with the Ctrl, Alt, or Shift keys. If the key value is an ASCII value, the Alt and Shift key combinations are not accepted. Instead, the defaults are `True` for Ctrl, but `False` for Alt and Shift. The Alt and Shift combinations can be used only with `VK_XXXX` (virtual key) combinations such as the Backspace (`VK_BACK`) or Delete (`VK_DELETE`) keys

Key Specifies the accelerator key. It can be one of the following:

- Integer, in the range 0 to 255. Integers are interpreted as ASCII or virtual-key values, depending on the `Type` property.

NOTE

Any single digit is interpreted as a key value. To enter an ASCII value from 0 to 9, precede the number with two zeros (for example, 006). In like fashion, two digit codes may be preceded with a single zero, although this is not a firm requirement.

- Character, for a character value. Optionally, the character value may be preceded by a caret (^) to signify a control character.
- Virtual-key identifier, for any virtual-key identifier. Select the desired `VK_XXXX` value from the drop-down list.

Type Identifies a key value as an ASCII value or a virtual key (`VirtKey`) value.

Next Key Typed Accepts the next key combination typed as the accelerator key. The `Key` and `Modifiers` values are changed to match. If possible, the key selected is always interpreted as a virtual key. The choice between entering an accelerator key definition directly or using the `Next Key Typed` option is purely a matter of personal preference.

String Resources

Treating strings as resources instead of scattering the strings throughout the program code is a distinct departure from conventional programming practices. Most compilers gather such static data together, usually positioning this data toward the end of the .EXE code, along with other static-data elements.

Defining strings as resources has three advantages:

- Like other resources, strings are loaded into memory only when and as they are needed.

- Strings in a string table can be modified more conveniently than strings scattered throughout the source code.
- Multiple string tables can be defined. Each table can provide different language versions and be loaded according to the user's preference.
- Standardization of error message syntax is more easily controlled when all strings are grouped in one location rather than scattered through multiple source files or even scattered within a single source file.

As with menu and accelerator key resources, string tables can be created using a plain-text editor, such as the Windows Notepad, Write, or Unipad (for non-English languages). Alternatively, you can create string tables using a resource editor.

String Resource Definition

String resources may consist of any type of string data and may be used for any of the same purposes as conventional string data, including window captions, messages, labels, or even brief explanations. (String table entries are generally not used for button and control captions or menus because these text strings can be handled through an image or menu editor.)

TIP

Previously, in 16-bit systems, string table entries were limited to a length of 255 characters. Today, in 32-bit systems, this limitation has been raised to a generous (and useful) 32KB. This means that you can put far more than simple messages in string tables—anything from .RTF and .HTML texts to custom-formatted messages. For examples of how string table entries can be put to new and expanded uses, see *Windows Error Messages*, also written by Ben Ezzell (published by O'Reilly & Associates).

Individually, each string definition follows C conventions and is enclosed in double quotation marks. Strings also accept C's special embedded characters, such as `\n` for a line feed, `\r` for a carriage return, `\t` for a tab character, `\\` for a single backslash, and `\"` for an embedded double-quotation mark.

String Table Construction



A typical (if brief) string table might look something like the following (this is excerpted from the FileView1.RC resource script):

```
STRING TABLE  
BEGIN
```



```
IDS_NAME,    "FileView"  
IDS_ERROR1,  "File size indeterminate"  
IDS_ERROR2,  "File too large for present example"  
END
```

Each string in a string table is identified by a short integer value, placing an upper limit of 65,535 strings in the string table. In this example, the string IDs are provided by constants defined in the `FileView.H` header.

In the second version of this string table, the actual values are substituted for the mnemonic constants, and there are a few additional strings.

```
STRING TABLE  
BEGIN  
    1, "FileView"  
    2, "File size indeterminate"  
    3, "File too large for present example"  
16, "this string belongs to another group"  
17, "together with this second string"  
32, "and a third group"  
END  
  
STRING TABLE  
BEGIN  
    44, "this string is defined in a second string table"  
    45, "as is this second string"  
    46, "and this third string"  
END
```

In this second example, two string table segments have been defined, but notice that there is nothing in the labels to identify these as separate segments.

More immediately important, strings are loaded in groups of 16, with all strings in a group loaded when any one of the strings is required. Groups are identified by their ID number, with numbers 0 through 15 forming the first group, 16 through 31 forming a second group, and so on. Thus, in the examples, strings 1, 2, and 3 form one group, strings 16 and 17 form a second group, string 32 is in a group by itself, and strings 44, 45 and 46 form a final group.

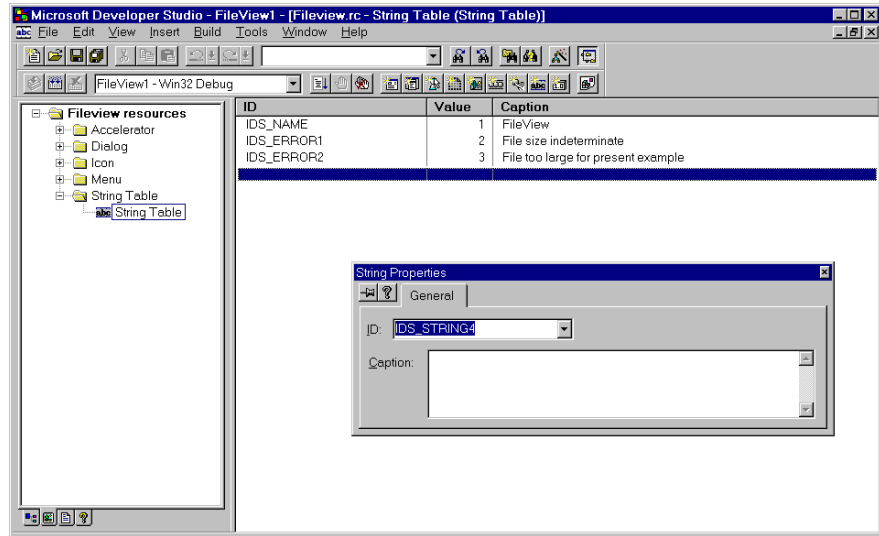
The programmer's objective is to group strings so strings that are needed will be loaded together, without loading unnecessary resources at the same time.

A String Table Editor

The Microsoft Developer Studio offers a separate string table editor, as shown in Figure S9.2.

FIGURE S9.2:

The Developer Studio string table editor



You can work in the Developer Studio string editor as follows:

- Add a string table entry by clicking the blank entry and entering a new ID and Caption in the Properties dialog box.
- Delete an individual string by selecting the string and pressing the Delete key.
- Move a string from one segment to another by changing the ID values.
- Move strings from one resource script (.RC) file to another by using the Cut and Paste options.
- Change a string or its identifier by editing the entry.
- Add formatting or special characters to a string.

Of course, you can accomplish the same tasks with a plain-text editor; the Developer Studio simply offers a more convenient tool for working with strings.

TIP

In the Developer Studio string editor, click the right mouse button to display a pop-up menu of resource-specific commands.

The string editor does not permit the creation of empty string tables. If you create a string table with no entries, it will be deleted automatically when you exit the Developer Studio.

Header File Resources

Header files provide a convenient place to define mnemonic constants used as links between resource files and application source code. It's far easier to remember that a radio button labeled `IDD_RES`, if selected, identifies a request for `.RES` source files than to remember that this button has a numeric identifier of 213.

Of course, once the `.RC` resource script has been compiled and linked with the application's executable code, the defined constants will all be replaced, but because the computer does lack a few of the programmer's more human shortcomings, this isn't the point. After all, the header file and the mnemonics are for the programmer's benefit, not the computer's.

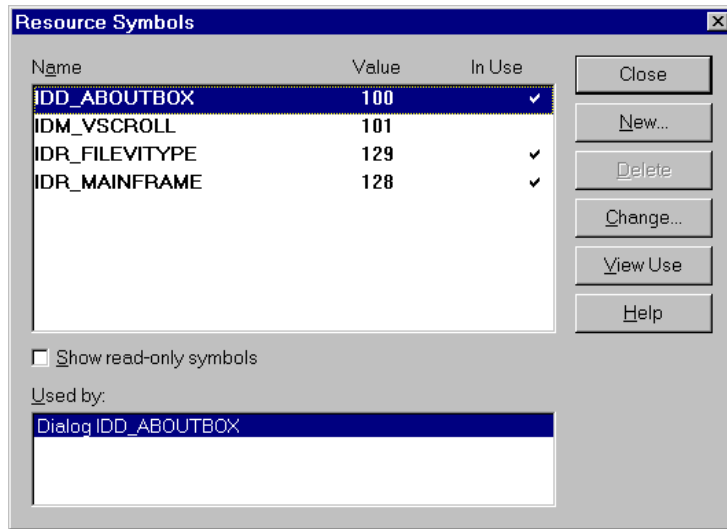
Like the other application resources discussed in this chapter, you can create header files using a plain-text editor. However, you can also use a resource editor's facilities for maintaining ID constants.

An Editor for Headers

To create headers using the Microsoft Developer Studio editor, choose Resource Symbols from the View menu to see the dialog box shown in Figure S9.3.

FIGURE S9.3:

The Developer Studio
Resource Symbols
dialog box



The Resource Symbols dialog box displays identifiers specific to the current project. It shows the value of the identifier, whether the identifier is used in the application (unchecked items are often orphaned IDs that you could remove), and where the selected ID is being used. If the ID is being used in more than one location, which is common, all of the uses will be listed.

When you select a use location and click the View Use button, the Developer Studio takes you directly to the appropriate source file.

The weakness of the Resource Symbols dialog box is that it does not function well with some non-MFC application source files. For example, the dialog box fails completely using the *FileView1* project, where the identifiers are all in the *FileView.H* header and not in a *Resource.H* header. However, in most circumstances, the Resource Symbols dialog box is a valuable tool.

TIP

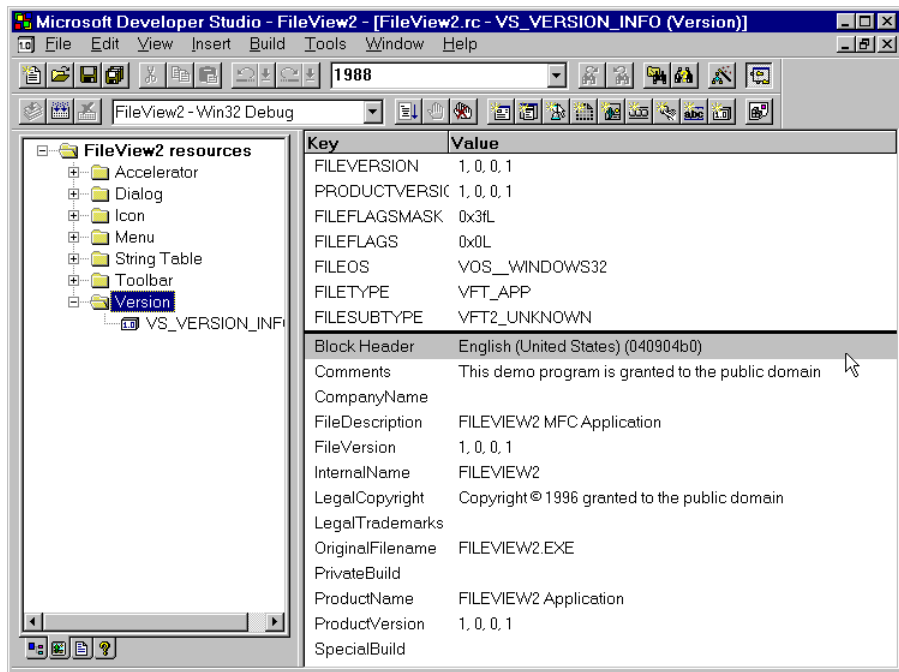
If you need to search for an identifier that is defined in one of the multitude of associated library headers, use the Find in Files button (or menu option) in the Developer Studio.

The Version Resource

The Version resource is a structured text block that contains company and product identification, a product release (version) number, and copyright and trademark notifications. The Version resource editor allows you to add or delete string blocks or to modify individual string values. Figure S9.4 shows the Microsoft Developer Studio version resource editor.

FIGURE S9.4:

The version resource editor in the Developer Studio



NOTE

The Windows standard is for an application to contain only one version resource under the name `VS_VERSION_INFO`.

To use version information within an application, the `GetFileVersionInfo` and `VerQueryValue` functions offer access to the file. While this version information is not required by any application, it is a convenient location to collect information identifying the application and version.

A Header File for FileView1

The following is the FileView.H header file, which is shown in text format, for the *FileView* demo (discussed in Chapter 12).

```
#define IDS_NAME      1
#define IDS_ERROR1    2
#define IDS_ERROR2    3

#define IDD_FNAME     16
#define IDD_FPATH     17
#define IDD_FLIST     18

#define IDM_QUIT      101
#define IDM_ABOUT     102
#define IDM_OPEN      103
#define IDM_TYPE      104

#define IDD_BMP       201
#define IDD_C         202
#define IDD_COM       203
#define IDD_CUR       204
#define IDD_DAT       205
#define IDD_DBS       206
#define IDD_DLG       207
#define IDD_DLL       208
#define IDD_EXE       209
#define IDD_H         210
#define IDD_PAL       211
#define IDD_RC        212
#define IDD_RES       213
#define IDD_TXT       214
#define IDD_ANY       215
```

This chapter completes our discussion of individual application resources. In the next chapter, we will look at the *FileView1* and *FileView2* demos, of which you've seen only fragments so far.

S U P P L E M E N T

T E N

S10

Application Resources Working Together

- Global application resources
- Three dialog box resources for the *FileView* application
- A common dialog box resource for Windows 98 applications

So far, we've discussed each of the individual application resources. You've seen fragments of the *FileView1* and *FileView2* demos as examples. In this chapter, we will focus on how all of the application resources work together in these two demos.

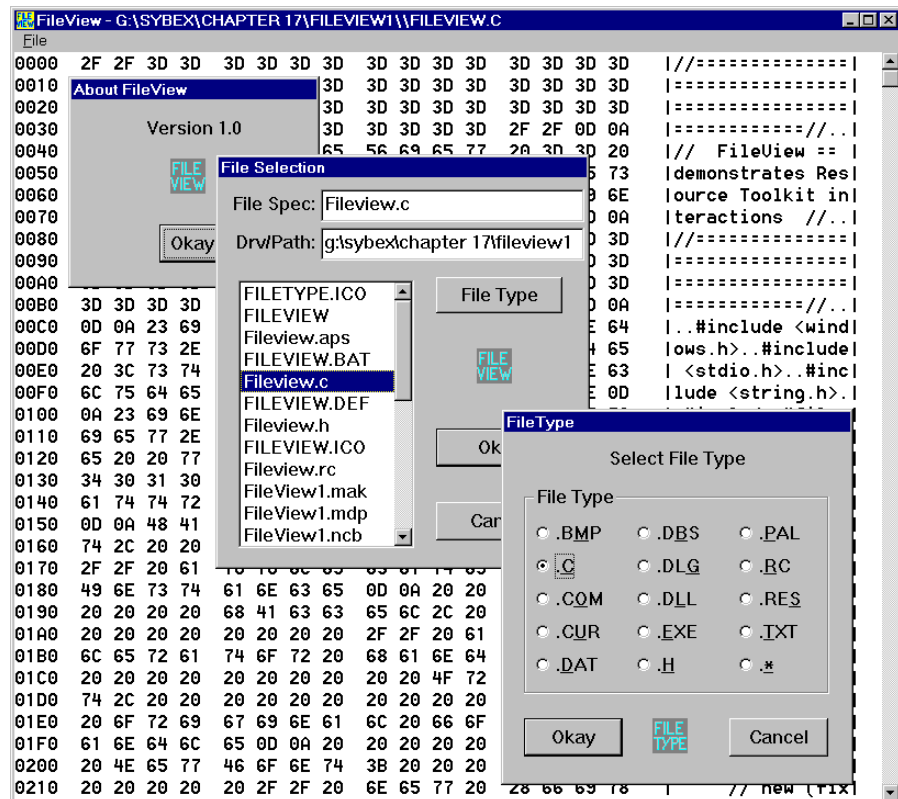
The FileView1 Demo: Using Application Resources



The *FileView1* demo is a relatively simple application designed to open a file of any type and display the contents in a columnar, hexadecimal format. It shows the file offset addresses at the right and the corresponding ASCII characters at the left. Figure S10.1 shows a composite of the application with its three dialog boxes.

FIGURE S10.1:

The completed
FileView1 demo



NOTE

The *FileView1* demo is included on the CD in the Supplement 10 folder.

WinMain Operations

As you know, Windows loads application resources only when and as they are required and discards them when they are no longer needed. However, in the *FileView1* demo, one resource is required during initialization of the first instance of *FileView*: the application title. This title is contained in the string table, not in the source code. Thus, if no previous instance of *FileView1* is active, the `LoadString` function is called during the instance initialization to retrieve the appropriate entry from the string table:

```
if( !hPrevInstance ) // if no prev instance, t's first
{
    LoadString( hInstance, IDS_NAME, (LPSTR) szAppName, 10 );
    wc.lpszClassName = szAppName;
```

Because the application title is global in this case, this string resource will be retained in memory without being discarded. In similar fashion, the application's cursor, icon, and menu are loaded as global resources:

```
wc.hCursor      = LoadCursor( NULL, IDC_ARROW );
wc.hIcon        = LoadIcon( hInstance, szAppName );
wc.lpszMenuName = (LPSTR) szAppName;
```

This case is not unique to the *FileView1* application; a similar set of assignments appears in all Windows applications. However, when using an MFC-based application, most of these operations are concealed from the programmer, as you may observe (or not observe) by checking the *FileView2* version.

Variable Initialization

Immediately following the instance initialization, provisions are also included to set default values for several global data variables:

```
iFileType = IDD_ANY - IDD_BMP;    // initial file type
lstrcpy( szFileExt, szFileType[iFileType] );
```

The `iFileType` variable is initialized using two symbolic constants, which are defined in the `FileView.H` header and which are used in the application resources.

Once this is done, the `szFileExt` variable is initialized to match. Notice, however, that only global variables are being initialized, and this does not affect the initial settings of the dialog boxes.

Keyboard Accelerator Loading

Later, but still in the `WinMain` procedure, another set of resources is required, because these, too, are global to the application and cannot be discarded until *FileView1* terminates. This resource is the accelerator resource set, which is loaded immediately after the application has been initialized:

```
hAccel = LoadAccelerators( hInst, "FILEVIEW" );
while( GetMessage( &msg, NULL, 0, 0 ) )
{
    if( !TranslateAccelerator( hWndMain, hAccel, &msg ) )
    {
        TranslateMessage( &msg );
        DispatchMessage( &msg );
    }
}
```

The `LoadAccelerators` function returns a handle to the accelerator resource table. However, because this handle is a global variable, the handle could be assigned to a different accelerator table elsewhere in the program as required, but it would still be used in the `TranslateAccelerator` call in the `WinMain` message loop.

The `TranslateAccelerator` function filters all of the keyboard-event messages directed to the application, permitting the majority of these to simply pass through without interference. However, when a key combination—or, more accurately, a key event accompanied by the appropriate shift-state flags—matches one of the resource accelerator-key events, this keyboard message is trapped or diverted. A new message—the action message defined for the accelerator event—is issued in its place.

Even applications that do not use accelerator keys will still call the message loop, but without the `TranslateAccelerator` invocation, as in:

```
while( GetMessage( &msg, NULL, 0, 0 ) )
{
    TranslateMessage( &msg );
    DispatchMessage( &msg );
}
```

Long Filenames

The *FileView1* demo can display complete long filenames. It is not limited to the older DOS-style 8.3 filename format, even though it uses only the standard Windows API functions.

For example, in the File Selection dialog box shown in Figure S10.1, you can see the long directory names chapter 17 and fileview1 in the Drv/Path edit box, and three filenames with more than eight characters appear in the file list.

If you browse around your hard drive using the *FileView1* demo, you should notice that all of your long filenames and long directory names—no matter how long they are—appear without any conversions to the archaic 8.3 format.

And, considering that this is accomplished without any special provisions in the code, you may wonder why so many applications that purport to be Windows NT/95/98 compatible or even “Designed for Windows NT/95/98” remain incapable of displaying anything except the old and obsolete 8.3 filename formats.

A certain amount of ire is a natural response as you go searching through a series of 30 or 40 subdirectories, each rendered as CHAP~nn, trying to find out where your Chapter 17 directory actually is. And, finally, if you are lucky and persevere, you may discover that Chapter 17 has been hashed to appear as CHAP~11— not exactly how you expected to find it.

Is this the product of a conspiracy of recalcitrant programmers who have banded together to attempt to preserve the old format in the same fashion that *Le Academy Francais* is attempting to preserve seventeenth-century French? Or is there some special stupidity at work here? It is definitely a mystery! (The solution appears later in this chapter.)

Dialog Box Handling

As discussed in Supplement 7, the *FileView1* demo uses three dialog boxes: About FileView, File Selection, and File Type. The three dialog boxes in the *FileView1* application were created as dialog box resources and are handled by procedures declared in `FileView.DEF` as exported.

NOTE

Exported procedures are simply procedures that need to be visible (accessible) from outside their immediate scope; that is, from outside the source file or class where they are declared. Procedures that are used only locally do not require export declarations. As a general rule, all functions that are used only within an EXE application are purely local. Functions within a DLL that are only called (used) by other functions and procedures belonging to the DLL are also local. This is true even for member functions—in either EXE applications or DLLs—belonging to one class that will be used by another class; as long as both classes exist in the same application; even if they are in multiple source files, they are still local functions. However, all functions within a DLL that will be called by applications using the DLL must be “exported” to make them accessible.

Once a dialog box has been invoked, the dialog box procedure exists as an effectively independent subprogram. The exported procedure receives its own messages from Windows, not directly from the parent application. Thus, declaring a procedure as exported makes the procedure’s address available to the Windows kernel so messages can be passed to the dialog box.

The three dialog box procedures are invoked from instructions in the `WndProc` procedure, with very minimal invocations, as:

```
case IDM_ABOUT:
    DialogBox( hInst, "ABOUT", hwnd, About );
    break;

case IDM_TYPE:
    DialogBox( hInst, "FILETYPE", hwnd, FileType );
    break;
```

Under Windows 3.1, this dialog box invocation would be somewhat more complex and would look something like this:

```
case IDM_ABOUT:
    lpProc = MakeProcInstance( About, hInst );
    DialogBox( hInst, "ABOUT", hwnd, lpProc );
    FreeProcInstance( lpProc );
    break;
```

In fact, this format is still acceptable under Windows 9x/2000, but it is unnecessary. The `MakeProcInstance` procedure from Windows 3.1 is now defined as a

macro that simply returns the first argument, with nothing required to create an instance of the procedure. And, because there is no requirement to create a procedure instance, the `FreeProcInstance` has also been redefined as a macro; this macro does absolutely nothing (aside from preventing the compiler from returning an error message because of an unrecognized term).

When the `IDM_OPEN` message is received, a slightly different response is used. It begins by initializing a directory path string and a file specification, passing these as arguments to the local `CallFileOpen` procedure, as:

```
case IDM_OPEN:      // set initial search path
    lstrcpy( szTmpFileSpec, szTmpFilePath );
    lstrcat( szTmpFileSpec, "*" );
    lstrcat( szTmpFileSpec, szFileType[iFileType] );
    if( CallFileOpen( hInst, hwnd,
        szTmpFileSpec, szFileType[iFileType],
        szTmpFilePath, szTmpFileName ) )
    {
```

The `CallFileOpen` procedure performs a few minor tasks of its own before using the same `DialogBox` API function, as was used to call the preceding two dialog boxes.

The `About` dialog procedure does very little except wait for the user to click the `Okay` button, after which it returns. There is no return value—or, at least, no response to a returned value—because nothing is decided in this dialog box.

The `FileType` and `FileOpen` dialog box procedures, however, are not as simple. Even though they return Boolean values, the major part of their work involves setting global string variables. This type of information cannot be conveniently treated as a returned value, even under Windows 9x/2000. However, the Boolean value returned by the exported `FileOpen` procedure to the `CallFileOpen` procedure and then to the `WndProc` procedure is used because there is no reason to attempt to open a file if the user hasn't selected one; that is, if the `Cancel` button was selected instead of the `Okay` button.

Before either of these procedures can return anything, the dialog box procedures themselves still have several tasks to perform, including initializing the dialog boxes before they are displayed.

Dialog Box Initialization

The `FileType` and `FileOpen` dialog box procedures handle a number of operations, but at the moment, their responses to the `WM_INITDIALOG` messages are the important topic.

The `FileType` dialog box procedure has only two important tasks. The first of these tasks is setting the check state of the radio button to match the `iFileType` variable, thus:

```
case WM_INITDIALOG:
    CheckRadioButton( hDlg, IDD_BMP, IDD_ANY,
                     IDD_BMP + iFileType );
    iInitType = iFileType;
    return( TRUE );
```

However, because the radio buttons used for the 15 file extensions were defined as auto radio buttons belonging to a single group, you are not required to reset the remaining 14; this part of the task is handled automatically.

This leaves the second task: setting the local variable `iInitType` so it's equal to the global `iFileType`. Once this is done, the `File Type` dialog box is ready for display and awaits the user's selection.

On the other hand, the `File Selection` dialog box is a bit more complicated to initialize. First, the `FileOpen` dialog box procedure retrieves the current or active drive and directory path and then sends the filename list box an initialization instruction setting the length of each entry to a generous 80 characters (just in case we want the long filenames supported by the NTFS file system or by the Windows 98 extended filename format).

```
case WM_INITDIALOG:
    GetCurrentDirectory( sizeof(OrgPath), OrgPath );
    SendDlgItemMessage( hDlg, IDD_FNAME, EM_LIMITTEXT, 80, 0L );
    DlgDirList( hDlg, szFileSpec, IDD_FLIST, IDD_FPATH,
               wFileAttr );
```

After initializing the list box, the next task is to fill it with entries from the current directory. The `DlgDirList` API function makes this task almost automatic. After all, aside from a handle to the dialog box (`hDlg`), the application is required

only to provide a file specification (`szFileSpec`), a destination (a list box identified as `IDD_FLIST`), a directory path specification (`IDD_FPATH`), and the desired file-attribute flags (`wFileAttr`).

In other circumstances, you might want to use two list boxes: one for filenames and the other for drive and directory information. This format is a convenient one used by many Windows applications. For this purpose, the `wFileAttr` flags would specify files for one list. For the other list, a `wDirAttr` flag variable would request directory information.

Finally, as a last step, the current file specification is added to the edit text box above the list box, as:

```
SetDlgItemText( hDlg, IDD_FNAME, szFileSpec );
return( TRUE );
```

Long Filenames: The Solution

The solution to limiting filenames and directory names to the obsolete 8.3 format is actually quite simple. By specifying a 12-character limit—8 for the filename, 1 for the dot, and 3 for the extension—in the `SendDlgItemMessage EM_LIMITTEXT` instruction, the system is forced to report only the old-style filename whenever a longer format filename is encountered.

```
case WM_INITDIALOG:
    GetCurrentDirectory( sizeof(OrgPath), OrgPath );
    SendDlgItemMessage( hDlg, IDD_FNAME, EM_LIMITTEXT, 12, 0L );
    DlgDirList( hDlg, szFileSpec,
                IDD_FLIST, IDD_FPATH, wFileAttr );
```

This limitation is definitely a stupid one and does not excuse programmers from recognizing (or failing to recognize) new realities.

The guilty parties—and you know who you are—are sentenced to 50 lashes with a wet data stream...and no excuses.

Conversely, by specifying a larger entry size—such as the 80 character size used in the `FileOpen` procedure—the system is allowed to report the new, long filename format.

Of course, in all fairness, if any of you are using filenames longer than 80 characters, the *FileView1* demo will truncate your filenames. But, somehow, I can't find it in my heart to worry unduly about this possibility.

Dialog Box Information Retrieval

In addition to setting initial information in the two dialog boxes, provisions are also required to retrieve information from them. And, as mentioned previously, each of these dialog box procedures is essentially an independent subprogram.

The FileType Procedure

In the `FileType` dialog box procedure, the important task is tracking the array of radio buttons; fortunately, this task is easily accomplished. Within the `FileType` dialog box procedure, button selections are tracked using the local variable, `iInitType`, leaving the global variable, `iFileType`, unaffected. In this fashion, when the user clicks any of the radio buttons, the array of buttons automatically resets because these were defined as auto radio buttons belonging to a single group. The selected button, however, sends an event message, which is intercepted, to the `FileType` dialog box procedure:

```
case WM_COMMAND:
    switch( LOWORD( wParam ) )
    {
        case IDD_BMP:    case IDD_C:
        case IDD_COM:    case IDD_CUR:
        case IDD_DAT:    case IDD_DBS:
        case IDD_DLG:    case IDD_DLL:
        case IDD_EXE:    case IDD_H:
        case IDD_PAL:    case IDD_RC:
        case IDD_RES:    case IDD_TXT:
        case IDD_ANY:
            iFileType = LOWORD( wParam ) - IDD_BMP;
            lstrcpy( szFileExt, szFileType[iFileType] );
            return( TRUE );
    }
```

Remember, the `wParam` argument contains the button identifier, which is a value that will be in the range of 201 to 215. The array of file-type extensions, however, has indexes from 0 to 14. Therefore, the constant `IDD_BMP` is subtracted from the low word in `wParam` to provide a usable index value.

Last, if the user clicks the `Okay` button, the global variable `iFileType` can be reset. If the user clicks the `Cancel` button instead, the global variable is left unchanged, despite any local selections.

The FileOpen Procedure

In the `FileOpen` dialog box procedure, the responses are not quite as simple. First, in addition to the `Okay` and `Cancel` buttons, a third button, labeled `File Type`, returns the command message `IDM_TYPE` if selected. This offers an alternative method of calling the `FileType` dialog box procedure from within the `FileOpen` dialog box procedure. The response provided is very similar to the provisions in the `WndProc` procedure:

```
case IDM_TYPE:
    if( DialogBox( hInst, "FILETYPE",
                  hDlg, FileType ) )
```

If `FileType` returns `TRUE`, meaning that a new file type was selected, the file specification is reset, and the `DlgDirList` function is called to update the directory list box, much the same as when the dialog box was initialized.

The list box, identified as `IDD_FLIST`, handles most of its own operations automatically, including scrolling, displaying text, and highlighting selections. There are, however, two messages that require handling within the dialog box procedure: the `LBN_SELCHANGE` and `LBN_DBLCLK` arguments accompanying an `IDD_FLIST` command message. These arguments are passed as high-word values in the `wParam` argument and must be tested using the `HIWORD` macro:

```
case IDD_FLIST:
    switch( HIWORD( wParam ) )
    {
        case LBN_SELCHANGE:
```

The `LBN_SELCHANGE` message simply states that a new item in the list box was selected and, in response, the edit file (`IDD_FNAME`) should be updated accordingly. As an alternative, if the dialog box had used a combo list box (combining a list box and edit box in a single feature), this task would be handled automatically, without involving the dialog box procedure.

The second `IDD_FLIST` argument, `LBN_DBLCLK`, states that an item in the list box has been double-clicked, indicating an immediate selection. In response, the first step is to call the `DlgDirSelectEx` API function to check if the selected entry is a directory or a file:

```
case LBN_DBLCLK:
    if( DlgDirSelectEx( hDlg, szFileName,
                      sizeof(szFileName), IDD_FLIST ) )
    {
```

```

        lstrcat( szFileName, szFileSpec );
        DlgDirList( hDlg, szFileName,
                    IDD_FLIST, IDD_FPATH,
                    wFileAttr );
        SetDlgItemText( hDlg, IDD_FNAME,
                        szFileSpec );
    }

```

If `DlgDirSelectEx` reports `TRUE`, meaning the selected entry is a new directory, the `DlgDirList` function is called to update the list box and the edit box, and the process proceeds as before.

On the other hand, if a `FALSE` result is reported, the selection must be a file-name rather than a directory and, therefore, the edit box is updated before sending an `IDOK` message to complete the selection process.

```

    else
    {
        SetDlgItemText( hDlg, IDD_FNAME,
                        szFileName );
        SendMessage( hDlg, WM_COMMAND,
                        IDOK, 0L );
    }
    return( TRUE );

```

Because the edit box offers the user a chance to type in an entry directly, the `IDD_FNAME` message is also checked for the `EN_CHANGE` argument in the high-word value of the `wParam` argument.

Two final provisions are responses for the `IDOK` and `IDCANCEL` command messages. In the case of an `IDOK` message, the response required is simply to check the edit box and retrieve the current entry before saving this as the selected filename.

```

case IDOK:
    GetDlgItemText( hDlg, IDD_FNAME, szFileName, 80 );
    ...
    EndDialog( hDlg, TRUE );
    return( TRUE );

```

The provisions for parsing the filename and the path/directory information are omitted here. The dialog box then closes with a return message TRUE to indicate to the calling procedure that a filename has been selected.

For the IDCANCEL message, the response is simple. After restoring the original drive/directory, the dialog box closes with a return message of FALSE to report that no selection has been made.

```
case IDCANCEL:
    SetCurrentDirectory( OrgPath );
    EndDialog( hDlg, FALSE );
    return( TRUE );
```

The *FileView2* Demo: Using a Common Dialog Box



The *FileView2* demo performs essentially the same task as *FileView1*, but instead of the File Type and File Selection dialog boxes, it uses the common File Open dialog box. Its main purpose is to demonstrate how an application can be revised to take advantage of the MFC libraries and of supplied resources.

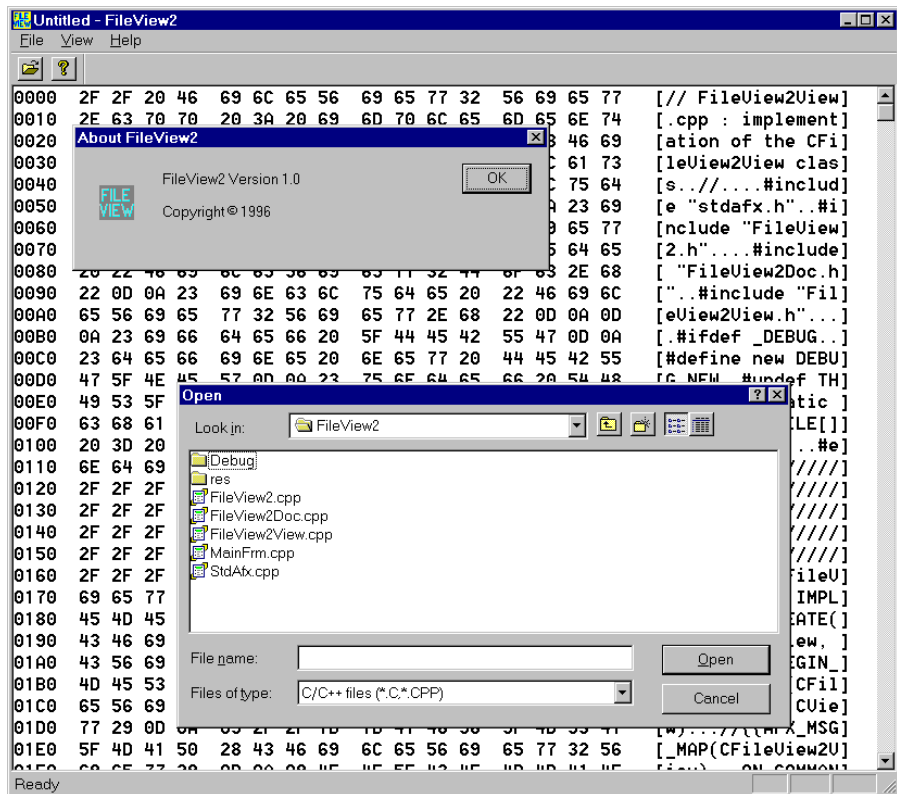
The *FileView2* version has only one resource dialog box, the About dialog box, because the File Open dialog box is a common dialog box resource supplied by Windows 98, not by the application. Also, where the *FileView1* demo employed a File Type dialog box, the *FileView2* version omits this resource by loading the list of file types and associated extensions in the File Open dialog box, where they appear in the Files of Type pull-down list. Figure S10.2 shows the *FileView* demo with its two dialog boxes.

NOTE

The *FileView2* demo is included on the CD in the Supplement 10 folder.

FIGURE S10.2:

The completed
FileView2 demo



A Pointer to the CFileDialog Class

The main reason for discussing this second version is to show how the File Type and File Selection dialog boxes were replaced by the common File Open dialog box. The heart of this part of the operation occurs in the `OnFileOpen` method in the `CFileView2View` class, where we begin by defining a pointer to the `CFileDialog` class.

```
void CFileView2View::OnFileOpen()
{
    // TODO: Add your command handler code here
    CFileDialog* pFileDlg;
    CString      csFilter;
```

The `CFileDialog` class encapsulates the Windows common File dialog box, which implements both the File Open and File Save As (or File Save) dialog boxes.

TIP

The File Open and File Save As dialog boxes can also serve for any other file-selection functions. Refer to the `CFileDialog` class documentation for details.

The Filter List

Defining a pointer to the class rather than an instance of the class provides a handle that will, in a moment, point to an instance of the class. Before creating an instance of the class, however, you must prepare.

```
csFilter =
    "Bitmaps (*.bmp)|*.bmp| "
    "C/C++ files (*.C,*.CPP)|*.c;*.cpp| "
    "Com files (*.com)|*.com| "
    "Cursors (*.cur)|*.cur| "
    "Data (*.dat)|*.dat| "
    "DBase files (*.dbs)|*.dbs| "
    "Dialogs (*.dlg)|*.dlg| "
    "Dynamic link libraries (*.dll)|*.dll| "
    "Executables (*.exe)|*.exe| "
    "Headers (*.h,*.hpp)|*.h;*.hpp| "
    "Palettes (*.pal)|*.pal| "
    "Resource scripts (*.rc)|*.rc| "
    "Resource files (*.res)|*.res| "
    "Text files (*.txt)|*.txt| "
    "All files (*.*)|*.*||";
```

The `csFilter` variable is an instance of the class `CString` and now contains a complete list of the file types and the file extensions for each type, with the `|` character used as a separator. Note also the doubled `||`, which terminates the string. When the `csFilter` string is passed to the `CFileDialog` instance, the `|` characters are interpreted as `0x00`—null characters that serve as delimiters.

Notice that each entry consists of two substrings. The first is the descriptive string, which will be displayed for selection. The second substring contains the file mask. In two cases—C/C++ and header files—two separate file masks are included by separating them with a semicolon (;).

There are no limits on the length of the prompt strings or file masks. Also, because a `CString` instance is not limited in length, there are no limitations on how many prompts and file masks can be included.

The only real stipulation that you need to observe is that the list provided will be presented in the exact same order; that is, the list of prompts and file masks will not be sorted.

The `CFileDialog` Instance

Now, once we have the filter list, the `CFileDialog` instance can be created thus:

```
pFileDlg = new CFileDialog( TRUE, NULL, NULL,  
                           OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,  
                           csFilter );
```

The calling parameters for `CFileDialog` are defined as:

```
CFileDialog( BOOL bOpenFileDialog, LPCTSTR lpszDefExt = NULL,  
            LPCTSTR lpszFileName = NULL,  
            DWORD dwFlags = OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,  
            LPCTSTR lpszFilter = NULL, CWnd* pParentWnd = NULL );
```

The parameters used are defined as follows:

bOpenFileDialog Set to `TRUE` to construct a File Open dialog box or to `FALSE` to construct a File Save As dialog box.

lpszDefExt A default filename extension. If the filename edit box entry does not include an extension when the File Open dialog box returns, the `lpszDefExt` extension will be appended to the filename automatically. If this parameter is `NULL`, no filename extension is appended.

lpszFileName A default filename that will appear in the filename edit box. If `NULL`, no initial filename appears.

dwFlags Flags that are used to customize the dialog box. For a complete list of flags and options, refer to the `OPENFILENAME` structure in the Win32 SDK documentation.

lpszFilter A sequence of string pairs that specify the filters that may be applied to the files. If no filter is supplied, all files are accepted, unless the user supplies a file mask.

pParentWnd A pointer to the parent or owner window. If no parent or owner is specified, the Desktop becomes the parent.

Having provided the appropriate parameters, the `CFileDialog` instance is called using the `DoModal` method.

```
if( pFileDialog->DoModal() == IDOK )  
{
```

Assuming that `DoModal` returns `TRUE`—meaning the user has selected the `Open` (`IDOK`) button rather than the `Cancel` button—the `GetPathName` method is called to retrieve the selected filename and the complete path/directory specification.

Note that this information is available after the dialog box returns but only as long as the `CFileDialog` instance has not closed. And, once you have this information, the `OpenFile` method is called.

```
    m_csFilePath = pFileDialog->GetPathName();  
    OpenFile();  
}
```

You can call the `OpenFile` method, which is a member of the `CFileView2View` class, without passing any arguments because the file information is contained in another member variable. This is the `CString` member `m_csFilePath`, which is directly available to the `OpenFile` method.

Last, call the `delete` operator to close the `CFileDialog` instance:

```
    delete pFileDialog;  
}
```

At this point, even though you're not finished with the file information (the file access and display is handled by other procedures), you are finished with the `CFileDialog` instance. Now you need to clean up to avoid a memory leak, which may be minor but can cause problems. In any case, all class instances should be closed when they are no longer needed.

WARNING

Memory leaks are very easy to cause and very, very hard to identify later when they begin causing major debugging headaches and other problems. This is an area where the proverbial ounce of prevention is worth far more than many pounds of cure.

The remainder of the *FileView2* demo provisions are quite similar to the ones in the *FileView1* demo. Its differences are the obvious changes appropriate to using an MFC-based application and some other changes to make appropriate use of object classes, such as *CStrings*, to replace more conventional variable types. If you are interested, take a look at the *OpenFile*, *FormatLine*, and *PaintFile* methods in the *CFileView2View* class.

TIP

Because the main point of *FileView2* is to demonstrate a few interactions between the program code and the application resources, this chapter has not explained all of that program's operations. Feel free to experiment with the source code listing (on the CD that accompanies this book), where you will find annotations to identify other areas of interest.

We've gone through the main types of application resources under Windows 98 and illustrated how they can be put together in your applications. In a sense, all of these elements are simply background—the nuts and bolts used for Windows applications. But, without the basic foundation, there's no point in trying to build more complete systems.

In the supplements that follow, we'll look at more advanced topics. We'll focus on creating tools and building simple applications to illustrate how the tools function.

Colors and Color Palettes

- The Windows standard palette
- How dithering works
- Custom color palettes
- Windows color drawing modes
- Color-to-gray-scale conversions

The procedures for handling colors under Windows are distinctly different from the color procedures under DOS. In part, this causes some additional complexity for the programmer, but overall, it provides greater flexibility. The application programmer no longer needs to make separate provisions for various hardware capabilities. Rather, you are free to devote your time to the application's principal objectives instead of writing code for a multitude of display systems.

For the application developer, one of the most important reasons to use color is to provide information in forms more readily recognizable than a simple monochrome display. Even for such simple tasks as spreadsheets or editing program code, the addition of color to highlight various elements provides additional, easily recognized information that could be lost on a simple black-and-white (or black-and-amber or black-and-green) display. With color, when an accountant says that a company is in the red, he can be speaking quite literally, without needing to reach for a red pen to make an entry in a ledger sheet.

In part, the use of color (or shades of gray) simply presents a display with the appearance of greater depth and detail than a monochrome display. And, even though first impressions are hardly an appropriate basis for judging an application's design and usefulness, they are still important. Your current and prospective clients will take that first look, and decide how much consideration to give to your application and how much effort they will expend in discovering the real strengths of your design.

Of course, there's also a flip side to this coin. Excessively elaborate graphics or absurdly garish color choices may have precisely the opposite of the desired effect: a bad first impression or, worse, general confusion while operating an application. Thus, restraint is also appropriate.

This is not, however, a lecture on the aesthetics of design (regardless of how relevant the topic might be). Instead, the present topic is how colors *can* be used. How you choose to use them remains up to you.

Windows Palettes

The earliest video cards provided video RAM (VRAM) on the order of 32KB to 64KB, enough for simple displays with limited colors. Today, even an inexpensive video card is expected to provide at least 1MB of RAM—enough for a 1024×768

display with a 256 color palette. And many video cards include twice this amount of RAM or more.

Color Definitions

Video uses an RGB color scheme to create colors. When painting a screen with light, combinations of red, green, and blue lights are sufficient to create an entire range of colors; black is the absence of any colored light, and white is a balanced combination of the three primary colors.

Using DOS, the video palette was defined in a relatively restrictive fashion with a fixed palette of 16 colors. These colors were defined using the RGBI flag system, where the flag bits controlled the red, green, and blue color guns together with a single intensity flag.

With the introduction of the EGA/VGA video boards, the palette expanded to 64 colors by exchanging the single intensity bit for three separate intensity bits: one each for the red, green, and blue color guns.

Today, both of these color specification systems have been supplanted by the Windows 32-bit color specification, where each color value is defined as a DWORD value in the format 0x00BBGGRR. In this format, the least-significant byte (eight bits) holds the value for red, the second byte is green, and the third byte is blue. The fourth, most-significant byte remains zero.

You may wonder why this is a 32-bit color specification when we’ve been talking about 24-bit color. The remaining eight bits in the DWORD value are used for a different purpose, which we’ll discuss later in the chapter.

Table S11.1 shows the CGA, EGA/VGA, and Windows equivalents for a 16-color palette.

TABLE S11.1: System Color Definitions

Color names	CGA Colors		EGA/VGA Colors		Windows Equivalents*
	binary	iRGB	binary	rgbRGB	
Black	0000	----	000000	0x00 00 00 00
Dark Blue	0001	...B	000001B	0x00 77 00 00

Continued on next page

TABLE S11.1 (CONTINUED): System Color Definitions

Color names	CGA Colors		EGA/VGA Colors		Windows Equivalents*
	binary	iRGB	binary	rgbRGB	0x .. BB GG RR
Dark Green	0010	..G.	000010G.	0x00 00 77 00
Dark Cyan	0011	..GB	000011GB	0x00 77 77 00
Dark Red	0100	.R..	000100	...R..	0x00 00 00 77
Dark Magenta	0101	.R.B	000101	...R.B	0x00 77 00 77
Brown	0110	.RG.	000110	...RG.	0x00 00 77 77
Light Gray	0111	.RGB	000111	...RGB	0x00 77 77 77
Dark Gray	1000	i...	111000	rgb...	0x00 3F 3F 3F
Light Blue	1001	i..B	111001	rgb..B	0x00 FF 00 00
Light Green	1010	i.G.	111010	rgb.G.	0x00 00 FF 00
Light Cyan	1011	i.GB	111011	rgb.GB	0x00 FF FF 00
Light Red	1100	iR..	111100	rgbR..	0x00 00 00 FF
Light Magenta	1101	iR.B	111101	rgbR.B	0x00 FF 00 FF
Yellow	1110	iRG.	111110	rgbRG.	0x00 00 FF FF
White	1111	iRGB	111111	rgbRGB	0x00 FF FF FF

*In a Windows color specification, the most-significant byte is used, in other circumstances, as a flag value indicating the type of color reference.

In the Windows color specification system, each primary color (red, green, or blue) has a possible range of 0 to 255, and individual colors are identified by 24-bit combinations of the RGB components, yielding a total of 16,777,216 possible hues.

However, because video boards (with the exception of the newest 24-bit video board) cannot support individual pixel color specifications, these 24-bit values are written to a color palette. The pixels in the image map itself consist of 8-bit index references to the color palette.

Thus, while an SVGA video card can support 24-bit color specifications, it can only do so as a palette containing 256 entries. Furthermore, partially in support of earlier 16-color standards, Windows reserves 20 of these palette entries, leaving 236 colors for custom use.

From the 16-Color to the True-Color Palette

Earlier EGA/VGA devices were limited to 16 colors, which were pixel colors defined by four color planes, each holding one bit per pixel. For the EGA/VGA drivers, the four color planes consisted of red (R), green (G), blue (B), and intensity (i). The intensity plane (or bit) shifted between green and light green, blue and light blue, and so forth. Thus, the 16 colors in the standard palette were determined by the available combinations of the three primary colors and the intensity of the color combination. White consisted of RGB and i, light gray was produced by RGB without the intensity bit, and dark gray was set by the intensity bit alone; that is, by very low levels of RGB combined.

For a VGA system, despite limitations inherited from the EGA color schema, there were actually six color planes consisting of both a high- and a low-intensity red (R and r), a high- and low-intensity green (G and g), and a high- and low-intensity blue (B and b). Thus, the VGA video card actually supported 64 (2^6) individual colors.

For the SVGA system, today's de facto standard, the color definition shifts from four or six color planes with one bit per pixel to one color plane holding eight bits per pixel, yielding a palette with 256 color entries (20 of which are reserved by the system for pre-defined hues).

Each of these palette entries is defined by three eight-bit values (RGB), giving each pure color a total of 256 levels (from black to full intensity) and a total of 16,777,216 possible colors (combining all possible RGB level combinations). Remember, however, that only 256 of these 16 million possible colors can be defined in the device palette, and the device—the video card—has only one palette.

For most purposes, a palette of 256 colors is adequate. Even my wallpaper and screen savers (which include paintings by Rousseau, wildlife and landscape photographs, and undersea images) are excellent when displayed in a 256-color palette. Of course, they are even more colorful when rendered using 24 bits per pixel color.

Generically titled *true color*, today's high-resolution video cards do not use palettes at all. Instead, they provide a full 24 bits of color information for each and every pixel in the display: eight bits each for the red, green, and blue components of every pixel.

Continued on next page

To provide this capability, the memory requirements rise drastically. For a 1024×768 display using 24-bit color, more than 2MB of VRAM is required. And, for a 1280×1024 display, the requirements jump to 4MB of VRAM.

But the VRAM requirements are only a part of the story. These upper-end video cards (as well as most of today's less expensive cards) use sophisticated graphics coprocessors, such as the Tseng ET-4000 or the newest S3 graphics coprocessors, because the amount of information required to render the screen display requires sophisticated processing as well as adequate storage. Without these coprocessors, rendering the screen display can slow the CPU to a crawl.

Ideally, true-color video boards provide enough memory to hold a full 24 bits of color. As a trade-off, however, some otherwise sophisticated video cards save memory by displaying only 16 bits per pixel rather than 24, using 5 bits each for the red, green, and blue components and using the sixteenth bit as an intensity flag applied equally to these components.

Because the human eye cannot distinguish between the 16- and 24-bit results—we simply cannot distinguish 16 million plus hues—the real requirements for high-resolution 24-bit color become somewhat problematic. Still, as memory costs fall and monitor sizes and resolutions rise, the overkill of full 24-bit color is the sunrise on the horizon.

But, even when every system sold is a true-color video system, there will still be a use for color palettes. The 256-color palette will be with us for a long time to come.

The Standard Palette

In DOS, the standard palette consisted of the 16 colors originally defined by EGA video cards or their VGA/SVGA equivalents.

Windows defines a standard palette of 20 static colors (the default palette). On an EGA or VGA system, where only 16 of the 20 colors are actually available, Windows emulates the remaining 4 colors by dithering (a process which will be demonstrated in a moment). On contemporary SVGA systems, where the hardware supports a device palette of 256 colors, the 20 default entries appear as individual hues without adjustments. Table S11.2 shows the RGB color values for each of the standard palette's 20 colors.

TABLE S 11.2: The Windows Default Palette Values

Index	Color	R	G	B	Index	Color	R	G	B
0	Black	0	0	0	10	Off-white	266	251	240
1	Dark Red	128	0	0	11	Med. Gray	160	160	164
2	Dark Green	0	128	0	12	Dark Gray	128	128	128
3	Gold	128	128	0	13	Red	255	0	0
4	Dark Blue	0	0	128	14	Green	0	255	0
5	Violet	128	0	128	15	Yellow	255	255	0
6	Dark Cyan	0	128	128	16	Blue	0	0	255
7	Light Gray	192	192	192	17	Magenta	255	0	255
8	Pale Green	192	220	192	18	Light Cyan	0	255	255
9	Pale Blue	166	202	240	19	White	255	255	255

Technically, these 20 standard colors belonging to the stock system palette are inviolable and cannot be altered by an application, even when the application defines its own color values for corresponding palette entries. Because color priority is given to the foreground application, however, and the application's palette takes priority over the standard palette, background displays may be remapped.

Background displays may appear in whichever application colors provide the closest match to the standard colors, even when this results in a distinct change in the screen appearance. In some cases, the color difference may be quite striking, such as when a 256-color bitmap is used as wallpaper and an application has defined its own 256-color palette. But once the new image is displayed, the background colors should return to their original palette colors. Similar effects can be observed using a paint program (such as PhotoShop Pro) when multiple images are loaded or while switching between images when a few moments are required to change between two quite different palettes.

NOTE

The *ViewPCX* demo, discussed in Supplement 15, provides a striking example of the effect of changing background display colors. As *ViewPCX* is loading an image from a file (having already defined a new palette), the background image is displayed using the *ViewPCX* palette.

The *Color1* Demo: Painting with the Standard Palette



The *Color1* demo was created to demonstrate the Windows standard color palette. As you can see in Figure S11.1, except for an optional icon, *Color1* has no menu, dialog boxes, or other resources. Execution occurs entirely within the exported *WndProc* procedure, with a minimum of operations.

FIGURE S11.1:

The *Colors1* display of the Windows standard palette



Because all that we intend to accomplish in the *Color1* demo is to paint squares using the default (stock) palette entries, we divide the client window into four rows of five squares each, defining these dimensions as *XSTEPS* and *YSTEPS*.

Within the *WndProc* procedure, in response to the *WM_SIZE* message, when we receive the client window size in the *lParam* argument, the *x* and *y* sizes for the individual rectangles are calculated.

```
case WM_SIZE:
    xSize = ( LOWORD( lParam ) ) / XSTEPS;
    ySize = ( HIWORD( lParam ) ) / YSTEPS;
```



```

        InvalidateRect( hwnd, NULL, TRUE );
        break;

```

Then, in response to the WM_PAINT message, we set up two loops:

```

case WM_PAINT:
    hdc = BeginPaint( hwnd, &ps );
    for( j=0; j<YSTEPS; j++ )
        for( i=0; i<XSTEPS; i++ )
        {

```

Within the loops, the first step is to use the PALETTEINDEX macro to convert the i and j values into a rectangle number and then, in the COLORREF crColor variable, into a palette index in the form 0x010000xx. PALETTEINDEX sets the high byte of the high word to 0x01, indicating that this COLORREF value is a palette-entry specification rather than an absolute RGB color value (see the next section, “Types of RGB Color Specifications”).

```

        crColor = PALETTEINDEX( i + ( j * xSteps ) );
        hPen = CreatePen( PS_SOLID, 1, crColor );
        hOldPen = SelectObject( hdc, hPen );
        hBrush = CreateSolidBrush( crColor );
        hOldBrush = SelectObject( hdc, hBrush );

```

Next, we need to create a pen and a brush using the crColor specification and then select the new pen and brush, making them the active drawing pen/brush for the next operations. Notice that we keep handles to the old pen and brush, supplied when SelectObject is called, so that we can reselect the original objects before deleting the two we created.

Once we have both a pen and brush in the desired color, we call the Rectangle function to draw a solid rectangle. The active (selected) pen is used to outline the rectangle and the active brush to fill it, but because both of these are the same color, we simply see a solid rectangle with no outline except the white background of the underlying window.

```

        Rectangle( hdc, i * xSize + 2, j * ySize + 2,
                   ( i + 1 ) * xSize - 2,
                   ( j + 1 ) * ySize - 2 );
        SelectObject( hdc, hOldPen );    // restore original pen
        SelectObject( hdc, hOldBrush ); // and brush then delete
        DeleteObject( hPen );            // discards so we don't
        DeleteObject( hBrush );          // run out of handles

```

Finally, as mentioned, we reselect the original pen and brush and then delete the new pen and brush. If we did not do this, we could run out of available handles.

TIP

To experiment, comment out the closing `SelectObject` and `DeleteObject` statements, then recompile and execute the demo. Then drag or resize the window—either of which results in redrawing the window. You should see the squares come up as white with a black outline once the custom brush and pen handles are exhausted. (The stock black pen and white brush remain available and are used by default when new handles are no longer available.)

Next, we call the `GetNearestColor` function, using the `crColor` palette-index specification, to get the actual RGB color values for the index entry. We report these, now using the default black pen and white brush, on top of the colored rectangles. The results are visible in Figure S11.1 (shown earlier).

```
        // report palette entries as RGB values
        crRGB = GetNearestColor( hdc, crColor );
        wsprintf( szColor, " RGB: %02X %02X %02X ",
                  GetRValue( crRGB ), GetGValue( crRGB ),
                  GetBValue( crRGB ) );
        TextOut( hdc, i * xSize + 20, j * ySize + 20,
                 szColor, strlen(szColor) );
    }
    EndPaint( hwnd, &ps );
```

This is a fairly simple process, but it does demonstrate the standard palette colors. It also leads to an explanation of three types of RGB color specifications.

NOTE

The complete listing for the *Color1* demo is included on the CD in the Supplement 11 folder.

Types of RGB Color Specifications

Three types of RGB color specifications are demonstrated in the *Color1* program: absolute, palette-index, and palette-relative values.

Absolute RGB COLORREF Values

Both the `CreatePen` and `CreateBrush` functions are called with a color reference parameter. Conventionally, this parameter is an RGB long integer (DWORD) following the form `0x00BBGGRR`, as shown in Table S11.1. Thus, for a white brush, the color value would be specified as `0x00FFFFFF`; for black brush, the value would be `0x00000000`.

In all cases, the most-significant byte is 0; the red, green, and blue values are each specified by byte (8-bit) values in the range 0 to 255. If necessary for display, the system will map the specified color to the nearest available color in the active palette.

Palette-Index RGB COLORREF Values

In the *Color1* demo, instead of using absolute RGB values, a second COLORREF format is used. In this format, the color parameter is a palette index identifying an existing palette value. For palette-index entries, the COLORREF value takes the form `0x0100xxxx`, with the low word (16 bits) providing an index to a logical palette or, in this case, the system palette.

Palette-Relative RGB COLORREF Values

Windows 98 also supports a third format for specifying COLORREF values: the palette-relative RGB value. For this format, the high-order byte value is 2 and the COLORREF value takes the format `0x02BBGGRR`. This format is used for output devices that support logical palettes, allowing Windows to match a palette-relative RGB value to the nearest actual color supported by the output device.

Alternatively, if the output device doesn't support a logical palette (probably because it is a video card supporting 24-bit true-color), Windows treats the palette-relative value as if it were an absolute RGB value; that is, instead of palette mapping, Windows attempts to handle the RGB value directly.

Dithered Colors

Although individual palette colors can be assigned to any of the 16 million possible hues, this does not guarantee that the physical device is capable of displaying such a wide range of colors. As explained previously, this limitation is imposed by the

graphics video card's limits more than by the video monitor's limits. Most monitors are capable of near-infinite color resolution.

Limitations imposed by the graphics hardware can be circumvented through a process known as *dithering*. The Sunday newspaper comic pages and comic books don't use the same precise techniques but the end results are very similar. In the comics and in colored ads, a fairly wide range of colors is produced by combining three or four primary colors to create what the eye perceives as many gradations of color.

The computer (or TV) screen creates colors by combining red, green, and blue light against a black background. Printed materials use a white background (the paper) and combine light-absorbing inks consisting of cyan, magenta, yellow, and black (CMYK). In this fashion, a strong brown, for example, is produced by placing dots of black or magenta and cyan over a nearly solid yellow background; a softer brown consists of a halftone of yellow with fewer blacks. Similarly, pinks and certain flesh tones combine yellow and magenta with the white paper showing through; dark colors use greater or lesser degrees of black.

On the computer screen, the same principle applies, except that the lights—the pixels—in primary colors are used rather than their complements as inks.

The **Color2** Demo: A Dithering Demonstration

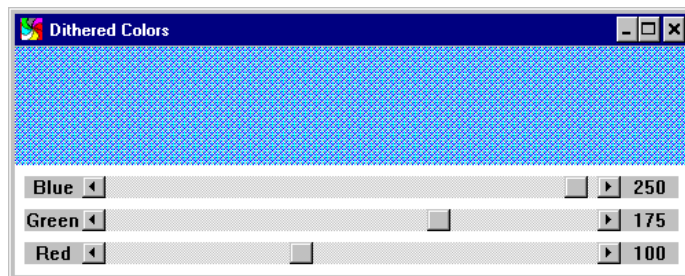


The *Color2* demo demonstrates this principle on an SVGA graphics system by setting the background color to a specific color configuration. Because the requested color is not provided as a palette color, Windows attempts to render the requested color by dithering entries from the default 20-color entries.

As shown in Figure S11.2, three scrollbars are used to select color settings for a single colored area, which displays dithered colors created from the standard palette.

FIGURE S11.2:

Creating dithered colors



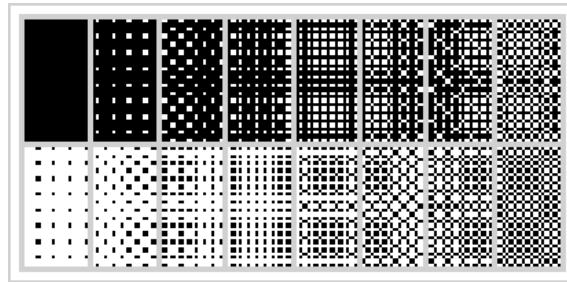
NOTE

The complete listing for the *Color2* demo is included on the CD accompanying this book, in the Supplement 11 folder. If your system is set for 24-bit color (true-color) or for anything greater than a 256-color palette, the *Color2* demo will render solid hues rather than dithered mixes. To execute the *Color2* demo, you must select a 256-color palette system.

Figure S11.3 shows a series of dithered color samples (unfortunately, rendered here in black and white).

FIGURE S11.3:

Dithered samples



Characteristics of Dithered Colors

Although dithered color patterns are a feature provided by Windows and do not require nor demand your attention, you should note the following characteristics:

- Dithered colors are always an 8×8 pattern, spreading the simulated color over a minimum area of 64 pixels.
- Dithering cannot be used for lines that are always drawn using a primary hue supported by the display device.
- Even though 20 (or more) individual hues are available, dithered colors are composed of 4 individual colors. (These are not, of course, the same 4 shades in all cases.)
- Although dithered colors will fill irregular outlines, individual pixels in the fill may combine (visually) with outlines or borders, creating some appearances of irregularity.

Dithering is not limited to color systems. It is also applied to monochrome and gray-scale displays, as will be discussed presently.

Custom Colors

In many cases, whether the resulting color specification appears as a solid hue or as a dithered pattern is irrelevant and makes no difference to your application. In other cases, however, precise color control can be a very important element. When this is the case, dithered colors just aren't in the running; it's either precise color control or nothing!

When exact colors are required, the solution is to reset one or more palette entries to produce the desired hues. Although this sounds simple enough, in practice, there are a few requirements and limitations.

The primary limitation is physical. Windows, no matter how sophisticated, cannot change the physical characteristics of the system video card. If the physical device supports a palette of only 16 colors, then only 16 custom colors can be displayed and all remaining palette entries will be mapped to the 16 supported physical colors. In like fashion, on an SVGA system, a physical palette limitation of 256 colors may be imposed by the system hardware.

Of course, if you are using one of the new true-color video cards that supports 24 bits of color information per pixel instead of palettes, all of this becomes moot, and you're free to write any information desired to the screen. This particular freedom, however, applies to very few programmers. Therefore, we'll assume that your programs are still bound by hardware limitations.

The *Color3* Demo: A Custom Color Palette



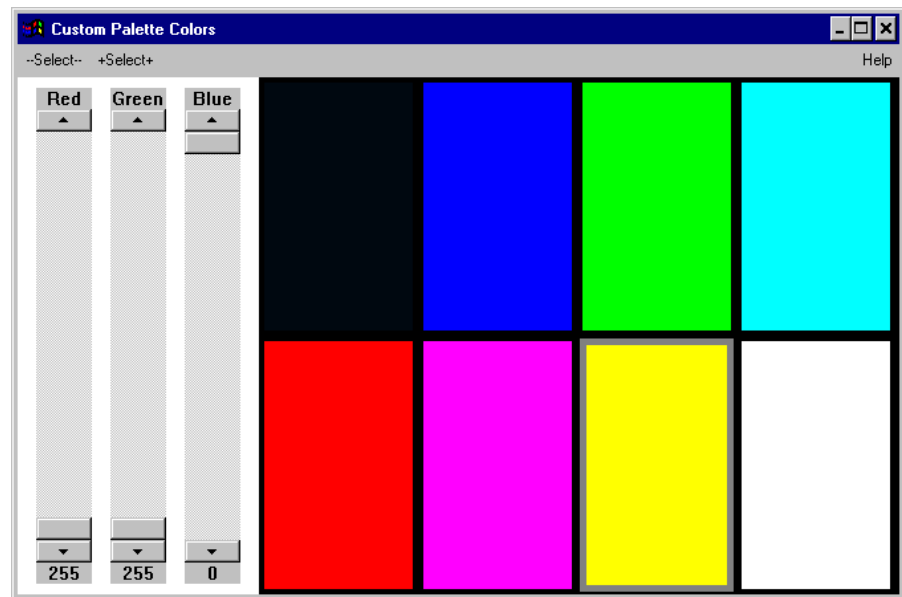
The *Color3* demo creates a new palette using custom color settings, which can be adjusted using the same scrollbar controls illustrated in *Color2*. Figure S11.4 shows the *Color3* display.

NOTE

For demonstration purposes, the custom palette used in this example is limited to eight entries. This is a range supported by all graphics cards. It's also large enough to compare several color samples but still small enough to present a clean display.

FIGURE S11.4:

The *Color3* display of a custom color palette



In the *Color3* demo, the three scrollbars control the red, green, and blue color specifications and also show the present levels for the selected color sample. The *-Select-* and *+Select+* menu items step through the eight palette entries, identifying the active selection with a gray outline.

The first step in creating a custom palette requires a few declarations:

```
long APIENTRY WndProc( ... )
{
    static LPLOGPALETTE 1Pal;
    ...
    HPALETTE      NewPal;
    HBRUSH        NewBrush, OldBrush;
    HPEN          NewPen,   OldPen;
    ...
}
```

The *1Pal* variable is a static pointer to a logical palette structure. The remaining variables provide handles (pointers) to two palettes, two brushes, and two pens, all of which will be used presently. Also defined, but not shown here, is an array of eight RGB color values that is used to initialize the color palette and to track changes in the color palette settings.

The logical palette structure (LOGPALETTE) referenced by `lPal` is defined in `WinGDI.H` as:

```
typedef struct tagLOGPALETTE
{
    WORD        palVersion;           // Windows version (0x0300)
    WORD        palNumEntries;        // size of array
    PALETTEENTRY palPalEntry[1];      // array of palette entries
} LOGPALETTE;
```

The version number is always 0x0300 (version 3.0), regardless of the version of Windows (3.1, 3.11, 95, 98, or NT) being used.

The `palPalEntry` field specifies an array of `PALETTEENTRY` data structures defining the actual color entries. The `PALETTEENTRY` structure is defined as:

```
typedef struct tagPALETTEENTRY
{
    BYTE peRed;
    BYTE peGreen;
    BYTE peBlue;
    BYTE peFlags;
} PALETTEENTRY;
```

The three color bytes accept values in the range 0 to 255. The `peFlags` field accepts a flag value identifying how the palette entry will be used, or it may be `NULL`. The following are the valid flag values:

PC_EXPLICIT Identifies the palette entry as a hardware palette index, allowing the application to use the display driver's palette. The RGB color specification will be used to find the nearest matching device palette entry.

PC_NOCOLLAPSE Specifies that the color will be placed in an unused entry in the system palette rather than being matched to an existing palette entry. If no unused entries are available, the color is matched normally. Once a new color entry has been made, further palette entries can be matched to this entry.

PC_RESERVED Indicates that the logical palette entry will be used for animation and, therefore, is changeable. As such, other palette entries should not be matched to this entry. If no unused palette entries are available for this color specification, the color specification is not matched to any other existing entries and is not available for animation.

If the `peFlags` field is `NULL`, the palette entry is added to the palette if space is available. If not, the entry is matched to the nearest system palette entry.

Much of the processing in *Color3* should be familiar to you from earlier examples in this and preceding chapters. The principal elements here are found in the `WndProc` procedure's message-handling provisions.

The initial color-processing provision is found in the response to the `WM_CREATE` message. Here, memory allocation is performed for the `lPal` palette structure and two initial values are assigned: the palette version and the number of palette entries.

```
switch( msg )
{
    case WM_CREATE:    // initialize the logical palette
        lPal = (LPLOGPALETTE)
            LocalAlloc( LMEM_FIXED | LMEM_ZEROINIT,
                sizeof(LOGPALETTE) + sizeof(PALETTEENTRY) * 8 );
        lPal->palVersion = 0x300;
        lPal->palNumEntries = 8;
        break;
```

Next, in response to the `WM_COMMAND` message, the `IDM_PLUS` (+Select+) and `IDM_MINUS` (-Select-) instructions step through the palette entries. As they step through the entries, they update the positions of the three scrollbars and the text displays for each to correspond to the current (active) palette settings.

```
case WM_COMMAND:
    switch( LOWORD( wParam ) )
    {
        case IDM_PLUS:
            nPal++;
            if( nPal >= 8 ) nPal = 0;
            for( i=0; i<3; i++ )
            {
                SetScrollPos( hwndScr1[i], SB_CTL, CVal[i][nPal], TRUE );
                SetWindowText( hwndVal[i], itoa( CVal[i][nPal], szBuff, 10 ) );
            }
            InvalidateRect( hwnd, 0L, TRUE );
            break;

        case IDM_MINUS:
            nPal--;
            if( nPal < 0 ) nPal = 7;
```

```

        for( i=0; i<3; i++ )
        {
            SetScrollPos( hwndScr1[i], SB_CTL, CVal[i][nPal], TRUE );
            SetWindowText( hwndVal[i], itoa( CVal[i][nPal], szBuff, 10 ) );
        }
        InvalidateRect( hwnd, OL, TRUE );
        break;

    case IDM_HELP: ...
}
break;

```

The scrollbars used for controls in both the *Color2* and *Color3* demos are the Windows analog of a vernier or sliding potentiometer control and should be familiar to you from many Windows applications. Their use here demonstrates how scrollbars can be used in any context where a variable control is needed.

TIP

Normally, to select a color specification, instead of placing scrollbars (or slider or other types of controls) in the main application or in a resource dialog box, the common dialog class `CColorDialog` is used.

Handling for the three scrollbars is found in three separate locations: one in the `WinMain` procedure, where the scrollbars are created, and two in the message responses in the `WndProc` procedure.

The first provision in the `WinMain` procedure, occurring at the same time the three scrollbars are created, is to assign a range to each scrollbar, consisting of a minimum and a maximum value. In each of the scrollbars used, the range assigned is 0 to 255, which is the range of the RGB color values. Initial values, or thumbpad positions, are also assigned at this time.

```

for( i=0; i<=2; i++ )
{
    hwndScr1[i] = CreateWindow( "scrollbar", OL, CHILD_STYLE | WS_TABSTOP | SBS_HORZ,
                               0, 0, 0, 0, hwnd, (HMENU) i, hInst, OL );
    hwndTag[i]  = CreateWindow( "static", szColorLabel[i], CHILD_STYLE | SS_CENTER,
                               0, 0, 0, 0, hwnd, (HMENU)(i+4), hInst, OL );
    hwndVal[i]  = CreateWindow( "static", itoa( CVal[i], szBuff, 10 ),
                               CHILD_STYLE | SS_CENTER,
                               0, 0, 0, 0, hwnd, (HMENU)(i+7), hInst, OL );
    SetScrollRange( hwndScr1[i], SB_CTL, 0, 255, 0 );
    SetScrollPos( hwndScr1[i], SB_CTL, CVal[i], 0 );
}

```

Next, in the response to the WM_SIZE message, the scrollbars are positioned within the application window and sized to fit appropriately.

```
case WM_SIZE:
    cxWnd = LOWORD( lParam );
    cyWnd = HIWORD( lParam );

    hdc = GetDC( hwnd );
    GetTextMetrics( hdc, &tm );
    cyChr = tm.tmHeight;
    cxChr = tm.tmAveCharWidth;
    ReleaseDC( hwnd, hdc );
    xOffset = cxChr * 26;

    xSize = ( cxWnd - xOffset ) / xSteps;
    ySize = cyWnd / ySteps;
    MoveWindow( hwndRect, 0, 0, cxChr * 26, cyWnd, TRUE );
    for( i=0; i<=2; i++ )
    {
        MoveWindow( hwndTag[i], cxChr * ( ( i * 8 ) + 2 ),
                    (INT)(cyChr * 0.5), cxChr * 6, cyChr, TRUE );
        MoveWindow( hwndVal[i], cxChr * ( ( i * 8 ) + 2 ),
                    cyWnd - (INT)( cyChr * 1.5 ),
                    cxChr * 6, cyChr, TRUE );
        MoveWindow( hwndScr1[i], cxChr * ( ( i * 8 ) + 2 ),
                    (INT)(cyChr * 1.5), cxChr * 6,
                    cyWnd - ( 3 * cyChr ), TRUE );
    }
    SetFocus( hwnd );
    break;
```

Last, in the response to the WM_HSCROLL message in the *Color2* demo (with horizontal scrollbars) or the WM_VSCROLL message in the *Color3* demo (with vertical scrollbars), the position of each scrollbar's thumbpad is adjusted according to where the scrollbar was clicked or where the thumbpad was dragged.

```
case WM_VSCROLL:
    i = GetWindowLong( (HWND) lParam, GWL_ID );
    switch( LOWORD( wParam ) )
    {
        case SB_PAGEDOWN:    CVal[i][nPal] += 15;                // no break!
        case SB_LINEDOWN:    CVal[i][nPal] = MIN( CVal[i][nPal] ); break;
        case SB_PAGEUP:      CVal[i][nPal] -= 15;                // no break!
        case SB_LINEUP:      CVal[i][nPal] = MAX( CVal[i][nPal] ); break;
```

```

        case SB_TOP:          CVal[i][nPal] = 0;          break;
        case SB_BOTTOM:      CVal[i][nPal] = 255;        break;
        case SB_THUMBPOSITION:
        case SB_THUMBTRACK:  CVal[i][nPal] = HIWORD( wParam ); break;
    }
    SetScrollPos( hwndScrl[i], SB_CTL, CVal[i][nPal], TRUE );
    SetWindowText( hwndVal[i], itoa( CVal[i][nPal], szBuff, 10 ) );
    InvalidateRect( hwnd, 0L, TRUE );
    break;

```

The `CVal` variable array consists of a 3x8 array of byte values containing the RGB color values for the eight palette entries used.

The real work of displaying the palette begins in response to the `WM_PAINT` message. It starts, as usual, with a `BeginPaint` instruction. However, before painting anything, a loop is used to read the present values from the `CVal` array into the palette entries indicated by `lPal`.

```

case WM_PAINT:
    hdc = BeginPaint( hwnd, &ps );
    for( i=0; i<8; i++ )
    {
        lPal->palPalEntry[i].peRed   = CVal[0][i];
        lPal->palPalEntry[i].peGreen = CVal[1][i];
        lPal->palPalEntry[i].peBlue  = CVal[2][i];
        lPal->palPalEntry[i].peFlags = PC_RESERVED;
    }
    NewPal = CreatePalette( lPal );
    SelectPalette( hdc, NewPal, FALSE );
    RealizePalette( hdc );

```

After initializing the palette values in memory, the `CreatePalette` function creates a new logical palette before `SelectPalette` makes `NewPal` the current (active) palette. Optionally, `SelectPalette` returns a handle to the old (default) palette.

Last, `RealizePalette` is called to activate the newly selected palette; that is, to make this the current drawing palette within the present device-context handle (`hdc`).

At this point, the device-context handle is ready for drawing using the new palette. The next segment of code consists of provisions to draw the eight rectangles composing the palette display.

```

for( i=0; i<8; i++ )
{
    j = i % 4;
    k = (int) i / 4;
    if( i == nPal )
        NewPen = CreatePen( PS_SOLID, 5, 0x007F7F7F );
    else
        NewPen = CreatePen( PS_SOLID, 1, PALETTEINDEX(i) );
    OldPen = SelectObject( hdc, NewPen );

    NewBrush = CreateSolidBrush( PALETTEINDEX(i) );
    OldBrush = SelectObject( hdc, NewBrush );

    Rectangle( hdc, xOffset + j * xSize, k * ySize,
               xOffset + ( j + 1 ) * xSize - 1,
               ( k + 1 ) * ySize - 1 );
}

```

Notice that within the loop, a parallel to the CreatePalette/SelectPalette provisions occurs as the CreatePen/CreateSolidBrush and SelectObject instructions create and select a pen and brush to draw each rectangle. The original (default) pen and brush have been saved as the OldPen and OldBrush handles.

```

SelectObject( hdc, OldBrush );
DeleteObject( NewBrush );
SelectObject( hdc, OldPen );
DeleteObject( NewPen );

```

After each pen and brush is used, the original pen and brush are reselected and the new pen and brush deleted.

Once the paint operation is completed, the new palette is deleted.

```

}
EndPaint( hwnd, &ps );
DeleteObject( NewPal );
break;

```

Each of these closing provisions is every bit as important as creating the palette, pens, and brushes in the first place. As stressed in Supplement 12, Windows can support only a finite number of handles to logical devices, so you must release handles when you no longer need them.

As a final provision, the new palette is deleted before the `WM_PAINT` message response concludes. The memory allocated for the palette structure and the pointer `lPal` is not released, however, and remains available for further use. This memory will be needed the next time the window is updated. All that has been lost is a temporary palette, a temporary pen, and a temporary brush. The originals have been restored, leaving the Windows system in the proper condition for other applications or for other actions by the present application.

WARNING

Remember, restoring the original condition is not just good manners—it's essential! If these handles are not released when they are no longer needed—immediately after use—and the originals restored, not only can the current application fail suddenly, but Windows itself can be left in a very hazardous state. If you wish to experiment, simply comment out the restoration provisions (but be sure to save your work before trying this).

There is one more element of cleanup required. This last bit is only necessary when the application exits and is handled in response to the `WM_DESTROY` message:

```
case WM_DESTROY:
    LocalFree( lPal );
    PostQuitMessage(0);
    break;
```

The `WM_DESTROY` message is an application's opportunity for a final cleanup before exiting. In previous examples, it has responded simply by posting a quit message to notify any child processes of an impending exit (a standard default provision even when there are no child processes). In this case, however, this is the appropriate point in time to release the memory allocated for the palette structure, as shown above, before notifying `WinMain`'s message loop to exit, completing the shutdown.

All special brushes and pens and the logical palette have already been taken care of within the paint procedure, and this concludes cleanup for the application.

NOTE

The complete listing for the *Color3* demo is included on the CD that accompanies this book, in the Supplement 11 folder.

Custom Brushes and Color Messages

In the *Color2* demo, a custom color was demonstrated by changing the background color. Then, in the *Color3* demo, custom colors were demonstrated by creating solid color brushes.

In both the *Color2* and *Color3* demos, an interesting addition would be to color the three scrollbars using the individual red, green, and blue settings. This is something you can try on your own. However, to facilitate the experiment, a few comments and suggestions follow.

In Windows 3.1, when a window control was to be redrawn, the parent window was sent a `WM_CTLCOLOR` message with the high word in the `lParam` argument containing the control type and the low word containing the control element's ID value. Both values, of course, were 16 bits.

Under Windows 9x/2000, where control elements are now 32-bit rather than 16-bit, the `WM_CTLCOLOR` message has been replaced by a series of seven `WM_CTLCOLORxxxxxx` messages, which explicitly identify the control element type, as shown in Table S11.3.

TABLE S11.3: CTLCOLOR Messages

Message Constant	Control Type
<code>WM_CTLCOLORMSGBOX</code>	Message box
<code>WM_CTLCOLOREDIT</code>	Edit control
<code>WM_CTLCOLORLISTBOX</code>	List box control
<code>WM_CTLCOLORBTN</code>	Button control
<code>WM_CTLCOLORDLG</code>	Dialog box
<code>WM_CTLCOLORSCROLLBAR</code>	Scrollbar control
<code>WM_CTLCOLORSTATIC</code>	Static control

Accompanying the `WM_CTLCOLORxxxxxx` message, the `wParam` argument contains a handle to the display context for the child window (the control to be repainted). The `lParam` argument contains the 32-bit child window handle.

There are a few cautions involved with using these messages. First, when an application explicitly processes any of these messages, the application must return a handle to a brush to paint the control background. If this is not done, the application will probably crash. A fragmentary example follows:

```
case WM_CTLCOLORxxxxxx:
    hCtrlBrush = GetWindowLong( lParam, GWL_ID );
    DeleteObject( hCtrlBrush );
    RGBColor = RGB( rVal, gVal, bVal );
    hCtrlBrush = CreateSolidBrush( RGBColor );
    UnrealizeObject( hCtrlBrush );
    return( hCtrlBrush );           // return the handle to the GDI
```

Another, less critical precaution, is to make sure that the application aligns the brush origin with the upper-left corner of the child window. If you don't accomplish this, particularly when you're using patterned brushes, the control may not be painted properly.

The MFC OnCtlColor Method

In applications using MFC classes, the corresponding operation is the `OnCtlColor` method, which is called when a child control is about to be drawn. Most controls send this message to their parent (usually a dialog box) to prepare the pDC for drawing the control using the correct colors.

In the `OnCtlColor` method, to change the text color used by a control, call the `SetTextColor` member function with the desired red, green, and blue values. To change the background color of a single-line edit control, the brush handle is set in both the `CTLCOLOR_EDIT` and `CTLCOLOR_MSGBOX` message codes. Also, in response to the `CTLCOLOR_EDIT` code, call the `CDC::SetBkColor` function.

Because the list box in a drop-down combo box is actually a child window belonging to the combo box but is not a child of the window, the `OnCtlColor` is not called for the list box. Thus, to change the color of the drop-down list box, create a custom `CComboBox` class that includes an override of `OnCtlColor` that checks for `CTLCOLOR_LISTBOX` in the `nCtlColor` parameter. In this handler, the `SetBkColor` member function must be used to set the background color for the text.

NOTE

The `OnCtlColor` member function is called by the framework to allow an application to handle a Windows message. The parameters passed to the function reflect the parameters received by the framework when the message was received. If the base-class implementation of this function is called, the implementation will use the parameters originally passed with the message and not any of the custom parameters supplied.

The `UnrealizeObject` Function

The `UnrealizeObject` function, called with a handle to an object, is used to reset the origin of the object. In the preceding example, the object was the handle to a brush that would be used to paint a control object's background. When you use `UnrealizeObject`, the GDI is directed to reset the origin of an object, such as a brush, when the object is next selected. The `UnrealizeObject` function is also used with logical palettes as an instruction to the GDI to remap the logical palette to the system palette.

The `UnrealizeObject` function should not be called while a drawing object, such as a brush or pen, is currently selected in a device context. However, when you use this function to remap a palette, the palette specified may be the currently selected palette in a device context.

The `DeleteObject` Function

During execution of the *Color2* demo, each time a `WM_HSCROLL` message is received, the existing background brush is deleted before a new brush (using the new color settings) is created. But before the application exits, a final call to the `DeleteObject` function is needed:

```
case WM_DESTROY:
    DeleteObject( (HGDIOBJ) GetClassLong( hwnd, GCL_HBRBACKGROUND ) );
    PostQuitMessage(0);
    break;
```

If we had included provisions to paint the scrollbars or other controls, these brushes would also need provisions to delete each object before exiting. In earlier examples, the only objects used were standard objects—brushes, pens, and the like—and so no special provisions for cleanup were needed. However, custom brushes, as well as other custom objects, require some memory. If you do not

delete custom objects prior to exit, they will continue to occupy memory (at least, until the computer is rebooted).

NOTE

An advantage in using C++ object classes, such as the MFC classes, is that class objects are self-destroying and delete themselves when they go out of scope, thus relieving the programmer of some of the cleanup tasks.

Color Drawing Modes

Under DOS, only one graphics drawing mode is supported. In this mode, each pixel drawn (including pixels comprising lines and the like) simply overwrites or replaces the existing pixels using the current drawing color.

In contrast, Windows supports multiple drawing modes, in which the image is combined with the existing (background) image in a variety of fashions. These drawing modes are referred to variously as *bit-wise Boolean operations* or, in Windows, as *raster operations*.

Windows ROP2 Operations

Since the drawing-mode operations involve two pixel patterns—the object image and the screen image—they are also referred to as *ROP2 operations*. In WinGDI.H, they are identified by R2_XXXX constants. Sixteen ROP2 operations are defined, as shown in Table S11.4.

TABLE S11.4: Binary Raster Operations

Mode Constant	Operation	Resulting Image
R2_NOP	Screen	Screen not affected (no operation)
R2_NOT	~Screen	Existing screen inverted
R2_COPYPEN	Pen	Pen overwrites screen (default)
R2_NOTCOPYPEN	~Pen	Inverted pen overwrites screen

Continued on next page

TABLE S11.4 (CONTINUED): Binary Raster Operations

Mode Constant	Operation	Resulting Image
R2_MASKPEN	Pen & Screen	Pen ANDed with screen
R2_MASKNOTPEN	~Pen & Screen	Inverted pen ANDed with screen
R2_MASKPENNOT	Pen & ~Screen	Pen ANDed with inverted screen
R2_NOTMASKPEN	~(Pen & Screen)	Pen ANDed with screen, result inverted
R2_MERGEPEN	Pen Screen	Pen ORed with screen
R2_MERGE NOTPEN	~Pen Screen	Inverted pen ORed with screen
R2_MERGE PENNOT	Pen ~Screen	Pen ORed with inverted screen
R2_NOTMERGE PEN	~(Pen Screen)	Pen ORed with screen, result inverted
R2_XORPEN	Pen ^ Screen	Pen XORed with screen
R2_NOTXORPEN	~(Pen ^ Screen)	Pen XORed with screen, result inverted
R2_BLACK	0	Black line (drawing color ignored)
R2_WHITE (R2_LAST)	1	White line (drawing color ignored)

The ROP2 constants listed are ordered according to function, not according to integer values. Thus, the first ROP2 mode listed, R2_NOP, has no effect on the screen image at all. However, it is still useful, since the current position (cp) is updated by LineTo or LineRel operations when using the R2_NOP mode.

The second ROP2 mode, R2_NOT, draws by inverting the existing image (for example, white becomes black) using bit-wise color inversion. This is useful for two reasons:

- It ensures absolute screen visibility (with the exception of screen areas that are approximately 50 percent gray).
- The original screen can be restored by executing a second, identical drawing operation.

The third ROP2 mode, R2_COPYPEN, is the default drawing mode, corresponding to the conventional DOS drawing mode discussed previously.

The next 11 ROP2 modes produce varying effects, which are more readily demonstrated by the *PenDraw1* demo than by description. The *PenDraw* demo is discussed in the next section.

The last two ROP2 modes, R2_BLACK and R2_WHITE, draw lines using complete black or white, respectively, regardless of the current drawing color.

The PenDraw1 Demo: Demonstrating Drawing Modes



The *PenDraw1* demo begins by writing labels along the right side of the screen before drawing a background. The background drawing starts at the left with five vertical gray bars, ranging from 0 percent black (white) to 100 percent black; then continues with six color bars in blue, green, cyan, red, magenta, and yellow.

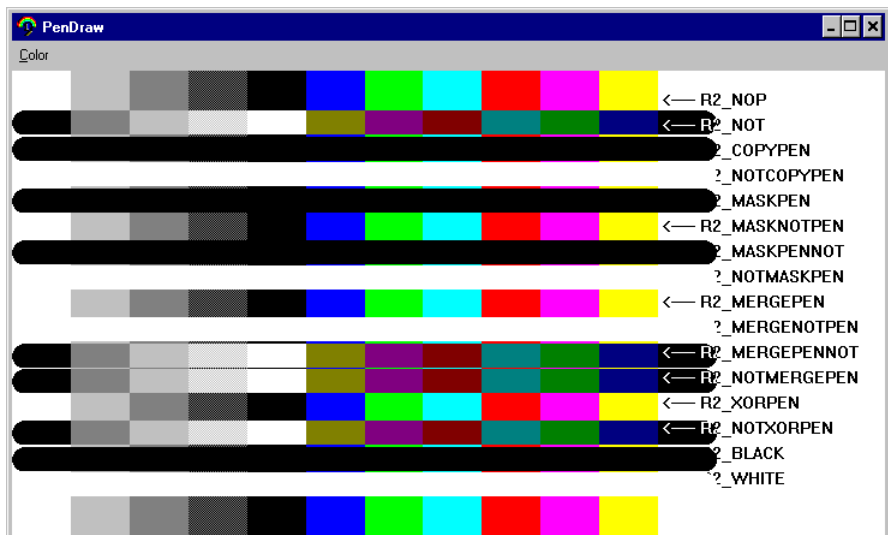
NOTE

The labels at the right in *PenDraw1* are intended to extend into the color bars at the left, further demonstrating how the ROP2_xxxx operations interact with background images.

Against this background, 16 horizontal lines are drawn using the 16 drawing modes (in the functional order listed in Table S11.4, not in numerical order), each employing the active drawing color. A range of eight drawing colors can be selected from the menu. Figure S11.5 shows an example of the *PenDraw1* display.

FIGURE S11.5:

Binary raster operations



Notice in Figure S11.5 and when executing the *PenDraw1* demo that the default white background is not treated in the same fashion as a white background drawn by a brush or as background to the text display. This is easily observed by selecting the yellow drawing color and observing the effects where the lines overlap the labels: R2_NOT, R2_MASKPEN, and R2_NOTXORPEN. Because the lines drawn are slightly wider than the text labels, a thin section of background white appears above and below the labels.

NOTE

The *PenDraw1* demo is included on the CD in the Supplement 11 folder.

Color to Gray-Scale

Windows has its own provisions for handling most gray-scale conversions for color programs executing on monochrome video systems. On some plasma and LCD screens (displays that are virtually an endangered species), even though the display is technically monochrome, the video system still accepts color-input information, translating the color data into 16, 32, or 64 gray levels.

In both of these situations, not only is the process of converting colors to gray-scale handled without the programmer's participation, but the programmer is effectively forbidden to intervene (except for offering palette choices that produce optimum contrast and clarity).

Even though most gray-scale conversions are handled by Windows, some circumstances do remain where programmers must supply their own conversions (for example, to show how a color image would appear on a monochrome printer, as explained in Supplement 17). How this is done depends on the circumstances, the equipment, and the desired results. There are no hard and fast rules, nor are there any absolutes.

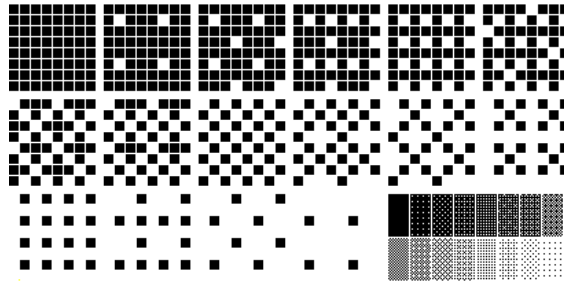
Following are a few suggestions for producing color to gray-scale conversions. You can apply these techniques to hard-copy devices, such as printers, as well as to monitors.

Gray-Scale Palettes

One popular method of accomplishing gray-scale conversion is to create a palette of grays suitable for mapping the original color palette. For example, assume a palette of 16 colors (as per EGA/VGA) ranging from white to black. The obvious gray scale for correspondence would be a 4×4 pixel (or dot) pattern, as shown in Figure S11.6.

FIGURE S11.6:

A 16-bit gray scale



Here, the 16-bit patterns range from solid black to one-sixteenth black. (Reduced views of the same patterns appear at the lower right.) As an alternative, you could drop one of the intermittent patterns to adjust the gray scale from solid black to solid white.

But remember, dithered colors use an 8×8 pattern. Applying similar patterns in black and white would offer a possible scale of 64 grays, providing a wider range of grays or a finer texture for hard-copy output.

WARNING

Adjacent elements in a 64-level, gray-scale palette can be very difficult to distinguish. Select carefully.

The gray-scale patterns suggested in Figure S11.6 form a uniform range that is about the best that can be accomplished with only 16 elements. Moving up to a 64-bit pattern presents the possibility of matching the gray-scale density to the intensity (or darkness) of the color being mapped.

To do so, the first step is to understand a few basic principles of color perception. The human eye does not perceive all colors equally, in terms of absolute intensity. Of the three primary colors—red, green, and blue—the eye perceives

green almost twice as strongly as red. In turn, the eye's response to blue is approximately one-third the response to red. Thus, an approximate gray-scale formula reflecting the perception curve of the human eye is:

$$\text{Intensity} = \text{Red} * 0.30 + \text{Green} * 0.59 + \text{Blue} * 0.11$$

Thus, since Windows uses the RGB values in the range 0 to 255, the gray equivalent matching a 24-bit color specification becomes:

$$W = \left(\frac{R * 0.30}{255} \right) + \left(\frac{G * 0.59}{255} \right) + \left(\frac{B * 0.11}{255} \right)$$

Using this formula, if the R, G, and B are all at maximum (255), resulting in white on the screen, the formula yields a value of 100 percent white and 0 percent black.

On the other hand, for a soft blue with an RGB value of 43, 128, 210, here's the formula:

$$W = \left(\frac{43 * 0.30}{255} \right) + \left(\frac{128 * 0.59}{255} \right) + \left(\frac{210 * 0.11}{255} \right)$$

and the equivalent proportions of white to black become:

$$W = (0.050) + (0.296) + (0.090) = 43.6\% \text{ white (or } 56.4\% \text{ black)}$$

Calculating a gray-scale using an 8x8 pattern, the optimum gray would be 36 black pixels (or dots) to 28 white pixels.

The problem in converting colors to grays is that quite distinct colors can yield the same gray values simply because they have the same relative intensities. For this problem, there is no simple cure.

On the other hand, optimizing printing a color image by creating 8x8 blocks for each pixel in the image is a rather frustrating process, if only in the annoyances involved in creating the 64-dot patterns need. Instead, there is a simpler approach.

Rather than creating an elaborate system of dots, you can simply convert the color image to a gray image by mapping the color pixels to their gray-intensity equivalents. For this, you can use the original formula:

$$I = (R * 0.30) + (G * 0.59) + (B * 0.11)$$

Once I has been calculated, the equivalent gray palette entry would be created as:

```
RGB( I, I, I )
```

Or, even easier, begin by creating a palette with gray-scale entries:

```
for( i=0; i<256; i++ )
{
    lPal->palPalEntry[i].peRed   = i;
    lPal->palPalEntry[i].peGreen = i;
    lPal->palPalEntry[i].peBlue  = i;
    lPal->palPalEntry[i].peFlags = PC_RESERVED;
}
NewPal = CreatePalette( lPal );
SelectPalette( hdc, NewPal, FALSE );
RealizePalette( hdc );
```

This code fragment would create and realize a palette with 256 shades of gray ranging from black to white. To convert a color image to gray, the only real requirement would be to calculate the intensity, using the perception-response formula, for each pixel and then assign the intensity as the pixel's palette index.

Once this is done, simply printing to any hard-copy device, which has its own routines to print gray equivalents, results in a black-and-white image that maintains the intensities of the original.

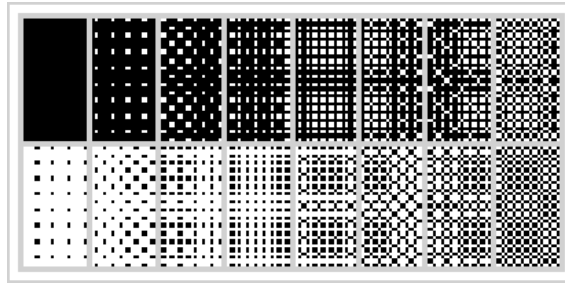
Gray Scales and Plaiding

There is one hazard inherent in using gray scale on black-and-white output devices: Plaiding can occur when the gray-scale pattern is not matched to the device resolution. Figure S11.7 shows an example of plaiding deliberately produced on the screen, showing that video devices are no more immune than printers to this problem. The illustration in Figure S11.7 is an excerpt from Figure S11.6, enlarged to show the mismatch between the image's dot pattern and the screen resolution.

As a simple rule of thumb, to prevent plaiding, make sure that the pixel dimension of the image (or dot dimension on a printer) after conversion to gray scale is an even multiple of the dot resolution of the reproduction size.

FIGURE S11.7:

Plaiding in gray-scales



For example, assume a 200×150 pixel bitmap is converted to a 16-shade gray scale. After conversion, the result is 800×600 pixels. To reproduce this image on a laser printer with a resolution of 300 dpi, without plaiding, the minimum size would be 2.666 inches wide by 2 inches high. Or, for a larger image, a print size of 8 inches wide by 6 inches high would also fit with the image scaled to 2400×1800 pixels.

Alternatively, if the image used a 64-shade gray scale, the smallest acceptable image would be 5.333 inches by 4 inches.

How you employ gray scaling is up to you and your requirements. Moreover, if you are content with the conversion capabilities provided by Windows and many output devices, you'll probably have little need for this facility. But if you do, you now know the basics of color-to-gray-scale conversion.

Gray Scale to Color Conversions

A less common requirement than color to gray scale is converting a gray scale to color. This process is commonly referred to as *false-color conversion*. Normally, this is not an attempt to reproduce a color image from a monochrome source, since there simply is not enough information for that task to be accomplished automatically. The automatic gray-to-color (false-color) conversion is an attempt to render an image where the only information is gray to a form where colors are used to make differences in intensity stand out.

Exotic examples include radio-star maps rendered in full glorious false color, or topological or meteorological maps, where color enhances readability. In infrared images, false-color assignments make it possible to print thermographic maps where temperatures are easily recognized as ranges of color.

TIP

The false-color process can be applied to virtually any type of information. There is just one thing that you must remember: The color information applied is purely arbitrary and should be chosen only for ease of recognition, not for artistic whim. Producing an image in alternating shades of chartreuse and puce may get you into the Guggenheim, but it's not a good way to convey information (unless you're trying to tell the world that you're color blind).

Implementation of false coloration is simple. Just decide on a color range and what levels of intensity to depict, and then construct a palette where the intensity levels (as palette indexes) have the appropriate color values.

Regardless of the number of levels in the original, it is often a good idea to restrict the false-color palette to a reasonably small number of hues, such as 20 or 32, rather than implementing a large palette of 256 shades. You should experiment to find the base palette size.

This chapter has covered all you need to know about handling color in your Windows 9x/2000 applications. We have discussed color palettes, custom colors, and gray-scale conversions. In the next chapter, we'll talk about the Windows drawing tools.

S U P P L E M E N T

T W E L V E

S12

Drawing Tools

- Line styles
- Hatch-fill styles
- Shape-drawing functions
- Business graphs: bar and pie charts

While folk wisdom maintains that a picture has value equal to a thousand words, this same adage has been most honored in dispute, disagreement, sarcastic rebuttal, and jest—not to mention outright subversion by pundits found everywhere from Madison Avenue to the halls of government. Still, the real truth might better be that, more often than not, a picture is preferred to a thousand words.

And, in like fashion, a graphic is often preferred to a thousand words. This preference, despite rumors concerning the literary acuity of CEOs and other board members, is not so much founded in any relative values but is based on the simple fact that a good graphic can convey information in a form more readily understood than many thousands of words or columns of figures.

One popular example of this principal is found in data-generated graphics in which images are created as visual analogs of numerical or scalar data, giving clarity to the relative relationships between elements at the expense of absolute magnitudes. Graphics of this type may be composed of simple shapes, such as those used in pie or bar graphs; may be less structured forms, such as with flow charts, schematics, or other diagrams; or may be composed of bitmapped images, as with the iconized buttons and controls found in any of a variety of Windows applications.

The topic of this chapter is creating graphics images using the drawing tools supplied by the Windows API functions. We will look at the various tools available, and then see how these tools work in applications.

Graphics Tools and Shapes

In Supplement 11, we discussed Windows color palettes and line-drawing modes. Those are the simplest of the tools supplied. Windows also offers a wide variety of other drawing features, including a selection of standard shapes, varying line styles, and a selection of fill styles for solid shapes.

Standard Shapes

Windows provides a series of functions to draw standard shapes, either as solids or outlines. Table S12.1 lists the functions and the shapes that they draw.

TABLE S12.1: Standard Shapes

Function	Shape
Arc	Open curve, either elliptical or circular
Chord	Arc with the endpoints connected by a chord
Ellipse	Closed curve, either elliptical or circular
Pie	Arc with endpoints connected to center
Polygon	Any multisided figure
PolygonPolygon	Multiple multisided figures
Rectangle	Rectangle with square corners
RoundRect	Rectangle with rounded corners

NOTE




The *PenDraw2* demo illustrates five of these eight shape functions, and the *PenDraw3* demo demonstrates two others. Both of these demos are discussed later in this chapter.

You'll see how to use these functions to create shapes presently. However, before these shapes can be drawn, a drawing pen is also required.

Logical Pens

Windows defines a selection of logical pens, each with a predefined pattern. The default pen if no other selection has been made is a solid, black line with a width of one logical unit. The defined pen (line) styles are listed in Table S12.2.

TABLE S12.2: Pen (Line) Styles

Style ID	Line Type
PS_SOLID	
PS_DASH	
PS_DOT	

Continued on next page

TABLE S12.2 CONTINUED: Pen (Line) Styles

Style ID	Line Type
PS_DASHDOT	— · — · — · — · — ·
PS_DASH2DOT	— · · — · · — · · — · ·
PS_NULL	No line (blank)
PS_INSIDEFRAME	If the pen width is greater than one logical unit, ensures that the line is drawn inside the closed shape. Valid with all primitive shapes except polygons.

NOTE

The PS_INSIDEFRAME style may be used in combination with any of the other line styles listed in Table S12.2. If the pen color does not match an available RGB palette color, the pen is drawn with a dithered (logical) color. If the pen width is one, PS_INSIDEFRAME is treated as PS_SOLID.

The initial step in selecting a new logical pen is to call the `CreatePen` function with specifications for the style, width, and drawing color.

```
hPen = CreatePen( nPenStyle, nPenWidth, RGBColor );
hOldPen = SelectObject( hdc, hPen );
```

After creating a new pen, the `SelectObject` function is called to associate the new pen with the device context, returning a handle to the previous pen.

Optionally, you could create a selection of pens—for example, as an array of handles—and then select each pen as needed (using `SelectObject`). But remember, each pen (or brush) you create consumes some memory. When the object is no longer needed, dispose of it via the `DeleteObject` function.

```
DeleteObject( hPen );
```

One caution: A created pen (or brush) should not be deleted while associated with a device context (unless, of course, the device context is about to be closed). Instead, before deleting a pen (or brush), the `SelectObject` function can be called to restore the original pen. For example, instead of simply calling `DeleteObject` with the handle of the pen to delete, use a compound statement:

```
DeleteObject( SelectObject( hdc, hOldPen ) );
```

Logical Brushes

Windows also defines a selection of logical brushes, each with a color specification and a predefined pattern. (Width, of course, does not apply.) Windows 98 provides a variety of hatched brushes (identifying constants are defined in WinGDI.H), which correspond to hatch-fill patterns supported by Windows 3.x. The hatch-fill styles are listed in Table S12.3.

TABLE S12.3: Hatch-Fill Patterns

Hatch Fill Style	Pattern
HS_HORIZONTAL	Horizontal lines
HS_VERTICAL	Vertical lines
HS_FDIAGONAL	Forward diagonal (forward slash marks, approximately 45°)
HS_BDIAGONAL	Backward diagonal (backslash marks, approximately 45°)
HS_CROSS	Horizontal cross-hatch
HS_DIAGCROSS	Diagonal cross-hatch

A logical hatch-fill brush is created in the same fashion as a logical pen, as explained in the previous section, and is subject to the same restrictions.

```
hBrush = CreateBrush( nHatchStyle, RGBColor );
hOldBrush = SelectObject( hdc, hBrush );
```

As with any other object, when you no longer need the brush, you should dispose of it.

```
DeleteObject( SelectObject( hdc, hOldBrush ) );
```

Guarding Against Squandered Memory

Unfortunately, you are allowed to create a brush or a pen without the formalities of saving a handle to either the new brush or pen, or the old brush or pen; that is, without making any provisions to delete the new object or restore the original. The following code will function without reporting any errors or warnings:

```
SelectObject( hdc, CreatePen( nPen-IDM_SOLID, 1, cColor ) );
SelectObject( hdc, CreateHatchBrush( nHatch-IDM_HORIZ, cColor ) );
```

Continued on next page

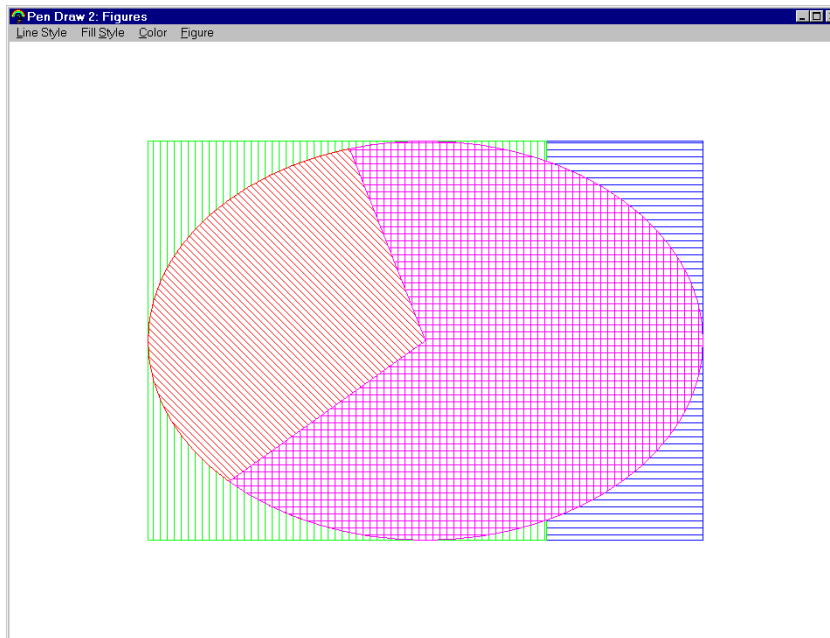
Each `CreateHatchBrush` and `CreatePen` call allocates memory for the brush or pen, which is not disposed of until either Windows is exited or the system is rebooted. The bottom line is simple: There are no guards against this type of error except for your awareness and careful programming practices.

The *PenDraw2* Demo: Drawing Shapes



The *PenDraw2* demo illustrates five of the eight shape-drawing functions: `Rectangle`, `Ellipse`, `Arc`, `Chord`, and `Pie`. This program permits you to select shape, line, and fill styles from a menu. The menu also offers a choice of colors, with a palette of eight shades predefined as RGB color values. Figure S12.1 shows an example of a shape drawn in *PenDraw2*.

FIGURE S12.1:



The *PenDraw2* demo

NOTE

The *PenDraw2* demo is included on the CD in the Supplement 12 folder.

Drawing Rectangles and Squares

The `Rectangle` function requires only four parameters to specify the coordinates (in device-context terms) for the upper-left and lower-right corners.

```
Rectangle( hdc, xUL, yUL, xLR, yLR );
```

A square is simply a special case of a rectangle and can be provided as:

```
Rectangle( hdc, xUL, yUL,  
          xUL + min( xLR-xUL, yLR-yUL ),  
          yUL + min( yLR-yUL, xLR-xUL ) );
```

The demo draws the rectangle or square using the current color, pen, line style, and fill style.

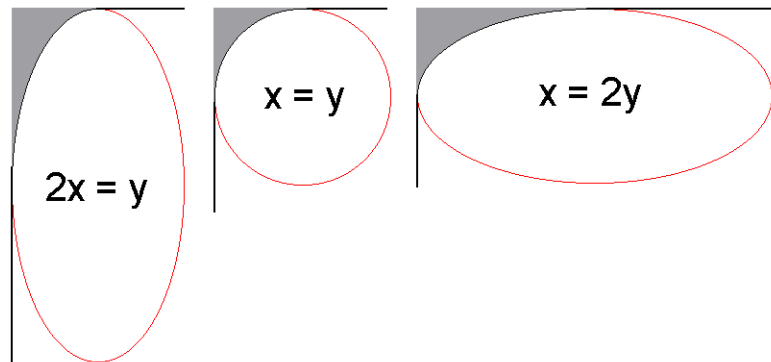
The `RoundRect` function (which is not demonstrated in *PenDraw2*) operates in the same fashion as the `Rectangle` function, except for the addition of two parameters specifying the x and y radii for the ellipsis forming the corners.

```
RoundRect( hdc, xUL, yUL, xLR, yLR, xRadius, yRadius );
```

In general, `xRadius` and `yRadius` are equal, making the corner arc circular, but this is not a fixed requirement; the corner ellipse can be elongated in either dimension. Figure S12.2 shows three corner examples: the left with `xRadius > yRadius`, the middle with `xRadius = yRadius`, and the right with `xRadius < yRadius`.

FIGURE S12.2:

Three corners using
`RoundRect`



Drawing Ellipses

In Windows, an ellipse is defined in terms of a theoretical rectangle bounding the ellipse. Like the `Rectangle` function, the `Ellipse` function is called with

four coordinates identifying the upper-left and lower-right corners of a bounding rectangle.

```
Ellipse( hdc, xUL, yUL, xLR, yLR );
```

NOTE

Mathematically and in more sophisticated applications, ellipses are described in terms of x and y radii and loci coordinates. In comparison, the C/C++ version of an ellipse is rather restricted in orientation.

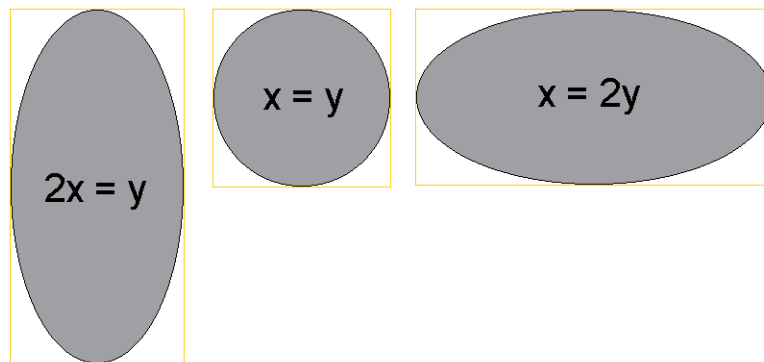
Also, just as a square is a special case of a rectangle, a circle is simply a special case of an ellipse, in which the x and y radii are equal.

```
Ellipse( hdc, xUL, yUL,  
        xUL + min( xLR - xUL, yLR - yUL ),  
        yUL + min( yLR - yUL, xLR - xUL ) );
```

Figure S12.3 shows three elliptical shapes together with their bounding rectangles. (These bounding rectangles are not drawn by the `Ellipse` function but are provided simply as illustration.)

FIGURE S12.3:

Three ellipses



Drawing Arcs, Chords, and Pies

Like the `Ellipse` function, the `Arc`, `Chord`, and `Pie` functions also use coordinate parameters to define a bounding rectangle that determines the shape of the arc, chord, or pie. But in addition to the basic curve, each of these functions also requires two pairs of additional coordinate parameters to identify the beginning

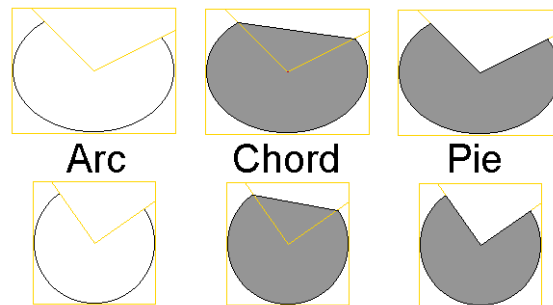
arc position (xp1,yp1) and the ending arc position (xp2,yp2). The three functions are called as:

```
Arc(   hdc, xUL, yUL, xLR, yLR, xStart, yStart, xEnd, yEnd );
Chord( hdc, xUL, yUL, xLR, yLR, xStart, yStart, xEnd, yEnd );
Pie(   hdc, xUL, yUL, xLR, yLR, xStart, yStart, xEnd, yEnd );
```

Figure S12.4 shows arc, chord, and pie shapes, together with the bounding rectangles and the radii determining the begin and end angles.

FIGURE S12.4:

Arc, chord, and pie shapes



NOTE

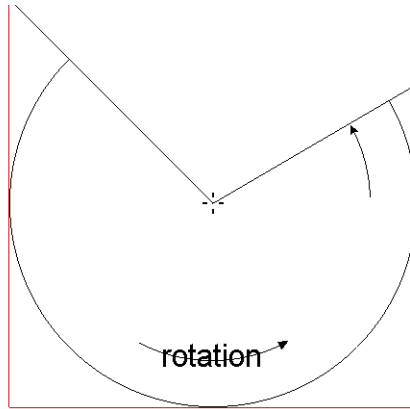
Under DOS, using C++ functions, an arc (or associated shape) is drawn by defining a center point, the radius (or x and y radii) and defining the beginning and endpoints as angles, with the 0° angle located horizontally to the right. In Windows, the shape of the arc segment, like the ellipse, is defined by a bounding rectangle.

The beginning and endpoints of arcs are defined, not by angles, but by points defining radii intersecting the arc. As shown in Figure S12.5, the arc is drawn counterclockwise, beginning at an angle defined by the xStart, yStart coordinates and ending at the angle defined by the xEnd, yEnd point.

The xStart, yStart point identifies a radius drawn from the center of the arc through the point specified and does not necessarily lie on the arc itself (though it may). The arc begins at the point where the radius and arc intersect. Or, if you prefer, the xStart, yStart point, together with the centerpoint, defines an angle for the arc starting point. In like fashion, the xEnd, yEnd point defines a radius setting the endpoint of the arc.

FIGURE S12.5:

Defining arc angles



For the **Arc** function, the process ends with determining the starting point and endpoint of the arc. For an arc, the resulting shape is not closed and no fill brush is used, although the arc itself is drawn using the current line style and color.

For the **Chord** function, the endpoints of the arc are connected with a straight line to complete a closed figure, which is filled in the *PenDraw2* demo using the selected hatch brush.

For the **Pie** function, the endpoints of the arc are also connected, but instead of a line between the two arc ends, two lines connect the endpoints with the center-point of the arc to create a pie slice. Again, the closed figure is filled using the selected hatch brush.

NOTE

Remember, for the **Chord** and **Pie** functions, the points used are the endpoints of the arc, not the points passed as arguments to define the radii, which, in turn, determine the arc endpoints. Later, in the *PieChart* demo, we'll use conventional trigonometry to calculate points that do lie on the arc (which is easier than calculating points that do not). Just keep in mind that these determining points are not required to lie on the arc itself.

Creating Business Graphs

One useful application for the shape-drawing functions is to create business graphs, such as bar graphs and pie charts. Although business graphs may not be your favorite subjects for programming (and certainly aren't mine), business applications are often required to produce such graphs. So, setting aside personal preferences, I've developed the *BarGraph* and *PieGraph* demos to illustrate how the *Rectangle* and *Pie* functions can be used with data sets.

Both the *BarGraph* and *PieGraph* demos use data arrays that are declared as static information within the program source code. In actual applications, of course, business graphs would use data either read directly from an external source or data calculated from external sources. For demo purposes, defining a data format and creating external source files is unnecessary for the actual objective. Do note, however, that both demos use the same data sets.

The *BarGraph* Demo: Building a Bar Graph



The *BarGraph* demo displays four years' worth of data broken down into eight categories. Colors identify data by years, and the bars are grouped by category.

TIP

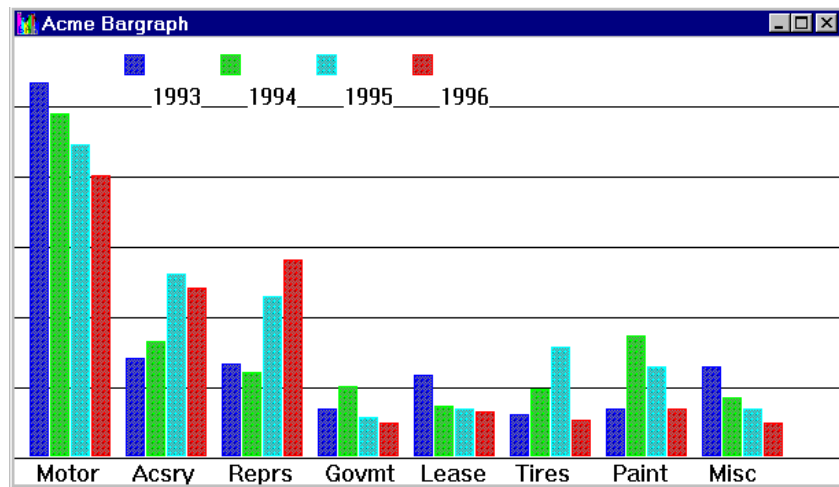
Optionally, you could also use varying fill patterns to identify category groups or to replace the year colors (for example, for monochrome displays).

In this application, there are advantages to using separate horizontal and vertical scale ranges and, therefore, it uses the `MM_ANISOTROPIC` mode. Another advantage of using anisotropic mapping is that it allows you to change the vertical scaling to accommodate variations in the maximum values that need to be graphed.

Once the mode is selected, the origin point is set near the lower-left corner of the window but slightly up and to the right, leaving room to accommodate labels below each group of bars. Also, after the client window is painted, the original (entry) mapping mode, which was saved when the `MM_ANISOTROPIC` mode was set, is restored, as are the original pen and brush sets. Figure S12.6 illustrates a sample bar graph.

FIGURE S12.6:

A sample bar graph



The principal elements specific to the *BarGraph* demo are found in the WM_PAINT response.

```
for( j=0; j<4; j++ )
{
    TextOut( hdc, ( j + 1 ) * 70 + 20, -2 * MaxVal - 20,
             szBuff, sprintf( szBuff, "%d", Years[j] ) );
    hPen = CreatePen( PS_SOLID, 1, lpColor[j+1] );
    SelectObject( hdc, hPen );
    hBrush = CreateSolidBrush( lpColor[j+1] );
    SelectObject( hdc, hBrush );
    Rectangle( hdc, (j+1)*70, 2*MaxVal+20, (j+1)*70+15, 2*MaxVal+5 );
}
```

The outer loop executes for the four years, writing a label to identify each year before creating a small block showing the color used for the year.

The next step executes a loop through the data elements for the year, creating a rectangle for each category using the brush and color created for the current year.

```
for( i=0; i<8; i++ )
    Rectangle( hdc, j * 15 + 1 + i * 70, 0, ( j + 1 ) * 15 + i * 70,
              2 * Accounts[j][i] );
DeleteObject( hPen );
DeleteObject( hBrush );
}
```

Last, the pen and brush objects are deleted because they are no longer necessary. However, compare the present usage to the methods suggested previously where the original pen and brush handles were saved and restored before the new pen and brush objects were deleted.

NOTE

The *BarGraph* demo is included on the CD that accompanies this book, in the Supplement 12 folder.

The PieGraph Demo: Building a Pie Graph



The *PieGraph* demo displays data for one year at a time in a pie-section format. Of course, pie graphs are generally expected to be round rather than elliptical. Instead of the MM_ANISOTROPIC mode, the *PieGraph* demo uses the MM_ISOTROPIC mode with the viewport origin in the center of the client window—a format selected for the convenience of the application.

Also, because C/C++ lack a predefined value for pi, PI2 is defined as a macro with the value $2.0 * 3.14159$, providing a means to convert values to angles (in radians) before using the derived angles to calculate points on the circumference.

The data used for the pie graph is an array of individual values. To draw the pie graph, these values must be converted into proportions of a total (proportions of the total circumference) before they can be converted to angles. Therefore, a loop is used to determine the total for the year:

```
TotVal[0] = 0;
for( i=0; i<8; i++ )
    TotVal[i+1] = TotVal[i] + Accounts[Year][i];
```

Once this has been done, the array TotVal contains the values necessary to calculate an angle for each category (in radians).

Before each pie section is calculated, a new pen and colored brush are created.

```
for( i=0; i<8; i++ )
{
    ...
    hPen = CreatePen( PS_SOLID, 1, lpColor[i] );
    SelectObject( hdc, hPen );
    hBrush = CreateSolidBrush( lpColor[i] );
    SelectObject( hdc, hBrush );
    ...
}
```

```

Pie( hdc, -Radius, Radius, Radius, -Radius,
    (int) ( Radius * cos( PI2 * TotVal[i] / TotVal[8] ) ),
    (int) ( Radius * sin( PI2 * TotVal[i] / TotVal[8] ) ),
    (int) ( Radius * cos( PI2 * TotVal[i+1] / TotVal[8] ) ),
    (int) ( Radius * sin( PI2 * TotVal[i+1] / TotVal[8] ) ) );

```

Because the mapping mode is isotropic and the viewport origin is at the center of the window, the rectangle bounding the pie section (or, more accurately, bounding the circle from which the pie section will be cut) requires no more calculation than the simple coordinate point pairs: $-Radius$, $Radius$ and $Radius$, $-Radius$. However, we do need to calculate the point coordinates for the starting and ending points for each pie section. For simplicity, these are calculated as points on the circumference of the pie.

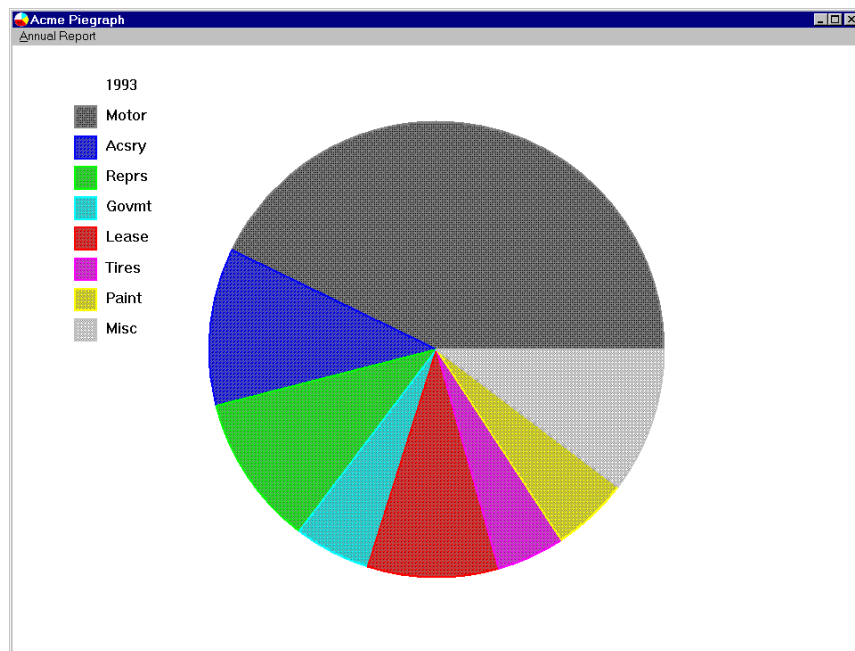
And that's it. Each pie section is created as a fraction of the total circle using a different pen color for the outline and a brush with the corresponding color for the interior. The results are shown in Figure S12.7.

NOTE

The *PieGraph* demo is included on the CD that accompanies this book, in the Supplement 12 folder.

FIGURE S12.7:

A sample pie graph



Drawing Polygon Figures

Like the `Rectangle`, `Ellipse`, and `Pie` functions, the `Polygon` and `PolyPolygon` functions draw bordered, closed, and filled shapes, but with a few differences. The first and principal difference is that the shapes can be more complex than a simple rectangle, although they are limited to straight lines; unlike the shapes drawn with the `Ellipse` or `Pie` function, the shapes drawn by either of the polygon functions cannot include curves.

The second difference is how you specify the data describing the shape. Where the `Rectangle` function expects a fixed set of coordinates, the `Polygon` function is more flexible and accepts a pointer to an array of coordinates (an array of `POINT`) with a further parameter specifying the number of points in the array.

```
Polygon( hdc, lpPoints, nPoints );
```

Each coordinate pair in the array of `POINT` identifies one vertex in a polygon. The `Polygon` function connects successive points with straight lines, finishing by connecting the last vertex to the first if necessary to close the shape. (For open shapes, use the `PolyLine` function.)

Similarly, the `PolyPolygon` function creates a series of closed polygons and is called as:

```
PolyPolygon( hdc, lpPoints, lpPolyCounts, nPolygons );
```

Again, the `lpPoints` parameter is a pointer to an array of `POINT`, identifying coordinates for each vertex in the polygon. The next parameter, `lpPolyCounts`, is a pointer to an array of integers, which defines the number of points in each polygon. The final argument, `nPolygons`, identifies the number of polygons (or, equally, the number of entries in `lpPolyCounts`).

Unlike the `Polygon` function, the `PolyPolygon` vertex arrays must be explicitly closed. The final vertex in each polygon must have the same coordinates as the first vertex, because `PolyPolygon` does not automatically close each figure. Also, using either function, individual polygons may overlap, but this is not required.

Polygon Fill Modes

The shapes we've used so far are simple, closed outlines with contiguous interiors, which did not require special handling to fill. However, the interior areas of polygon shapes may or may not be contiguous; if the area is not contiguous, it requires a different approach for filling.

For this reason, two different fill modes are supported (the names describe the algorithms used to determine which points lie inside the figure and which lie outside):

Alternate This fill mode considers regions as interior only when they are reached by crossing an odd number of boundaries (1, 3, 5, and so on). Regions reached by crossing an even number of boundaries are not filled.

Winding This fill mode, although slower to calculate, has the advantage of filling all interior (bounded) regions irrespective of the number of boundaries crossed.

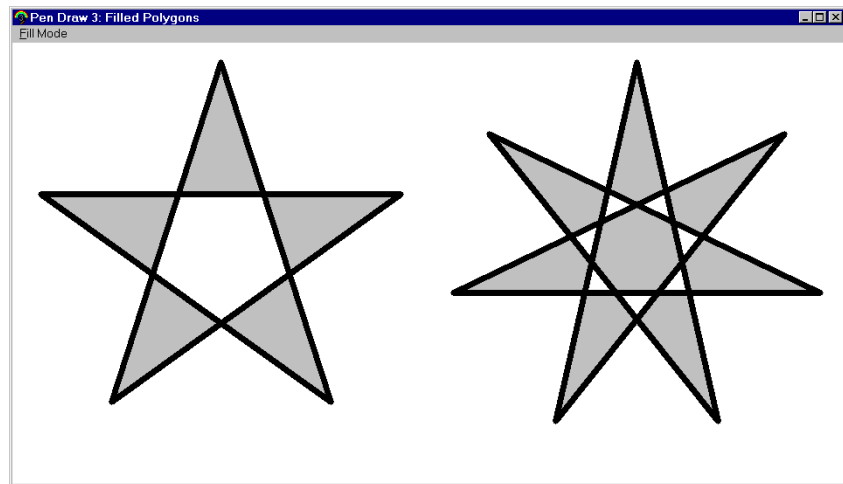
The *PenDraw3* Demo: Creating Polygons



The *PenDraw3* demo demonstrates the `Polygon` function by drawing two polygons: a five-pointed star and a seven-pointed star, as shown in Figure S12.8. In the figure, each shape has been filled using the alternate algorithm. The winding algorithm is available as a menu selection and will fill all interior spaces.

FIGURE S12.8:

Polygons and fill modes



The demo draws the shapes after calculating the appropriate vertices, using simple trigonometric functions similar to those employed in the *PieGraph* demo. For the five-pointed star, the vertex coordinates are calculated in the order 0, 2, 4, 1, 3, using the formula $j=(j+2)\%5$. (If these points were calculated in successive order, the result would be a simple pentagon with a contiguous interior.)

```

for( i=j=0; i<5; i++, j=(j+2)%5 )    // 5 points
{
    pt[0][i].x = (int)( sin( j*PI2/5 ) * 100 ) - 110;
    pt[0][i].y = (int)( cos( j*PI2/5 ) * 100 );
}

```

For the seven-pointed star, the formula $j=(j+3)\%7$ serves the same purpose, with the points calculated in the order 0, 3, 6, 2, 6, 1, 4.

```

for( i=j=0; i<7; i++, j=(j+3)%7 )    // 7 points
{
    pt[1][i].x = (int)( sin( j*PI2/7 ) * 100 ) + 110;
    pt[1][i].y = (int)( cos( j*PI2/7 ) * 100 );
}

```

The constants used—+110 and -110—offset each star to the right of center and the left of center.

Alternatively, to use the `PolyPolygon` function, instead of calculating the points for the two shapes, you could use a static array of points:

```

static POINT pts[] =
{
    -110, 100, -52, -80, -205, 30, -15, 30,
    -168, -80, -110, 100, 110, 100, 153, -90,
    32, 63, 207, -22, 13, -22, 188, 62,
    67, -90, 110, 100 };
static int poly[] = { 6, 8 };

```

The array `pts` provides the vertexes for the two shapes, and the array `poly` declares the number of points in each polygon. With this data available, the `PolyPolygon` function could be called as:

```

PolyPolygon( hdc, &pt, &poly,
            sizeof(poly) / sizeof(POINT) );

```

Remember, where the `Polygon` function for our example requires only five and seven vertex coordinate points, respectively, the `PolyPolygon` function requires six and eight vertex coordinate pairs. The final coordinate points in each set are the same as the first, thus closing each figure.

NOTE

The *PenDraw3* demo is included on the CD in the Supplement 12 folder.

We've covered four graphics elements in this chapter: pen styles, fill patterns, drawing functions for regular shapes, and drawing functions for irregular shapes. These are only a few of the graphics functions supported by Windows, and they are also the simplest. More sophisticated graphics functions are demonstrated in the following chapters, beginning in Supplement 13 with bitmap graphics operations.

S U P P L E M E N T

T H I R T E E N

S13

Brushes and Bitmaps

- Data-array defined bitmaps
- Resource bitmaps
- Old-style bitmaps
- Device-independent bitmaps

In Supplement 12, you learned how to use a variety of solid and hatched (or patterned) brushes to fill shapes. In this chapter, you'll learn that any pattern (bitmap), within certain limitations, can be used as a brush pattern.

Of course, fill patterns are only one of many uses for bitmaps. Because bitmapped brushes provide both a beginning point and one of the simplest uses, we'll start our coverage of bitmaps with this subject. Later chapters will cover more complex uses of bitmaps.

Bitmaps Defined as Arrays

While obvious to the point of being trite, the first step in creating a bitmapped brush is creating the bitmap itself. For a brush, this will be a (minimum) 8×8 bitmap image.

You could define a bitmap image within the source code as an array of BYTE, for example:

```
static BYTE wBricks[] = { 0xFF, 0x08, 0x08, 0x08, 0xFF, 0x80, 0x80, 0x80 };
```

This array defines an 8×8 bit pattern similar to the BRICKS image shown in Figure S13.1, which appears a bit later in the chapter, and could be used to produce a pattern brush similar to the one shown in Figure S13.4 (left half of pentagonal star), also presented later in the chapter. However, notice that the preceding statement has been qualified using the condition "similar." Even though the patterns are similar, the array `wBricks` describes a monochrome pattern, while the brush patterns used in the two illustrations are polychrome.

Array to Bitmap Conversion

In order to use `wBricks` as a brush pattern, the next step is to call the API function `CreateBitmap` to convert the value array into a (memory) bitmap image:

```
hBitmap = CreateBitmap( 8, 8, 1, 1, (LPSTR) wBricks );
```

The `CreateBitmap` function creates a device-dependent bitmap (in memory) for monochrome images. The parameters work as follows:

- The first two parameters (8, 8) are the width and height specifications.
- The third parameter (1) sets the number of color planes in the bitmap (each plane has `nWidth * nHeight/nBitCount` bits).

- The fourth parameter (1) sets the number of color bits per display pixel. (Remember that `wBricks` describes a monochrome image pattern, which is compatible with all video systems.)
- The final parameter is a pointer to the array of bytes, which defines the initial bitmap bits. If this argument is `NULL`, the bitmap will remain uninitialized.

After you create a device-dependent bitmap, the next step is to create a brush using the pattern.

A Brush with the Bitmap Pattern

To create a brush using the bitmap pattern, you use the bitmap handle with a call to `CreatePatternBrush`:

```
hBrush = CreatePatternBrush( hBitmap );  
SelectObject( hdc, hBrush );
```

After calling `SelectObject`, the new pattern becomes the current brush object.

Don't forget that you should delete both the bitmap and the brush when they are no longer needed:

```
DeleteObject( hBrush );  
DeleteObject( hBitmap );
```

Disadvantages of Bitmaps Defined as Arrays

Even though bitmaps can be defined as arrays of data within the program source code, this approach has three principal drawbacks:

- With the exception of monochrome images, the bitmaps created using `CreateBitmap` are device-dependent. Thus a bitmap defined for VGA, using four color planes with one color bit per pixel (per plane), will not be compatible with an SVGA system that uses a quite different arrangement.
- Although monochrome bitmaps can be written out as hex data, color bitmaps in the same format are a real pain to create.
- Static data arrays within the compiled application waste memory during execution—and do so quite unnecessarily. Granted, under Windows 9x/2000, this may be less of a problem than with previous Windows versions, but why bother when there are simpler ways?

Resource Bitmaps

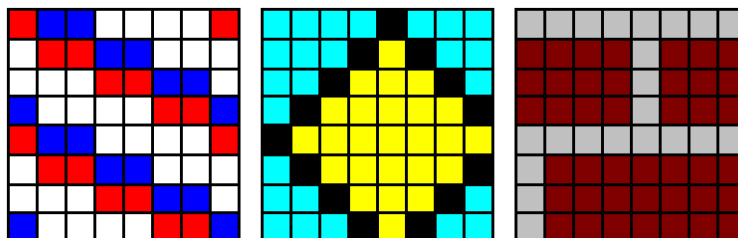
Resource bitmaps offer an alternative to bitmaps defined as arrays, without the problems. This approach has the following advantages:

- Using a bitmap editor (or any other paint program) makes creating bitmaps convenient, regardless of whether they are monochrome or color.
- The bitmaps created are device-independent, whether in monochrome or color, and can be displayed on any video system. Windows supplies any necessary conversions.
- Since the bitmap data is contained in the resource section and loaded into active memory only as needed, the data does not waste memory when not required.

For our example of how to use resource bitmaps, we'll use four bitmap images. The first three are for the stripes, diamond, and bricks brushes, as shown in Figure S13.1. The fourth is the chains bitmap, shown in Figure S13.2.

FIGURE S13.1:

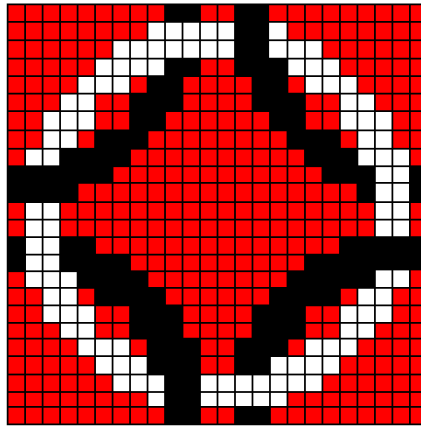
Three 8×8 bitmap patterns



The three bitmaps in Figure S13.1 are acceptable for use as brushes under Windows 98 (as well as under Windows 95/NT/2000). However, under Windows 98 (or Windows 95 or Windows 3.1), if the bitmap pattern is larger than 8 pixels square, only the upper-left corner (8×8) of the image is used. Under Windows NT/2000, there is no limit on bitmap brush size.

FIGURE S13.2:

A 24×24 bitmap pattern



This discrepancy between Windows NT/2000 and Windows 98 can be seen in Figure S13.3, where the same bitmap (24×24) results in quite different brush patterns. When you execute the *PenDraw4* demo (discussed next) under Windows 98 and select the chains bitmap, the resulting brush fill pattern will look like the one shown on the left side of Figure S13.3. When the application is executed under Windows NT/2000 and the chains bitmap is selected, the resulting fill pattern uses the entire bitmap image, as shown on the right side in Figure S13.3.

FIGURE S13.3:

The chains bitmap under
Windows 98 and NT/2000

Image in
Windows 98



Image in
Windows NT/2000

After the bitmap images have been created, either as part of a .RES resource file or as external .BMP images referenced by a .RC resource script, the linker combines these with the rest of the resources as a part of the .EXE executable. Remember that the resource section of the application is not loaded on execution. Instead, elements from the resource section are loaded on demand as required and discarded when no longer required.

The *PenDraw4* Demo: Using Resource Bitmaps for Brushes



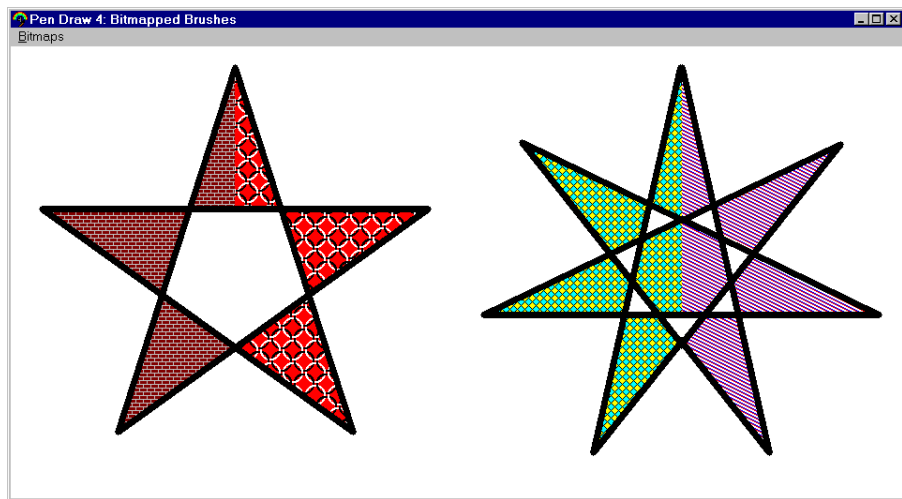
The *PenDraw4* demo uses four bitmaps to create four patterned brushes, which are selected from the menu. The brushes are used to draw the same five- and seven-pointed stars that the *PenDraw3* demo creates, as described in Supplement 12.

Besides the resource bitmaps used for the brushes, there is one other important difference between the *PenDraw3* and *PenDraw4* demos. In *PenDraw3*, the two figures are created from calculated data using the `Polygon` function. In *PenDraw4*, a single static array of coordinates is used together with the `PolyPolygon` function, which was discussed but not demonstrated in Supplement 12.

The *PenDraw4* demo uses only one brush at a time. However, for illustration purposes, Figure S13.4 shows a composite of the four bitmapped brushes.

FIGURE S13.4:

Four patterned brushes for the *PenDraw4* program



NOTE

The *PenDraw4* demo is included on the CD in the Supplement 13 folder.

Loading the Bitmaps

Using these bitmaps as resources begins with a `LoadBitmap` instruction. In the *PenDraw4* demo, this is accomplished in the exported `WndProc` procedure in response to a `WM_CREATE` message:

```
static HBITMAP hBitmap[4];
...
switch( msg )
{
    case WM_CREATE:
        hBitmap[0] = LoadBitmap( hInst, "BRICKS" );
        hBitmap[1] = LoadBitmap( hInst, "CHAINS" );
        hBitmap[2] = LoadBitmap( hInst, "DIAMOND" );
        hBitmap[3] = LoadBitmap( hInst, "STRIPES" );
        break;
```

The four bitmap images are loaded using a static array of handles (`hBitmap[]`). The `LoadBitmap` function loads the bitmap resource specified (by the `lpBitmapName` argument) from the resource section of the executable or, optionally, from some other module specified by the `hInst` parameter.

TIP

Alternatively, you could load individual bitmaps as they are selected, load them globally from the `WinMain` procedure, or, in another application, load them when some subprocedure is initiated.

Creating and Selecting the Brush

After loading a bitmap, the next steps are to create the brush and select the brush as the active object. In *PenDraw4*, these tasks are executed in response to the `WM_PAINT` message.

```
hBrush = CreatePatternBrush( hBitmap[ nBitmap ] );
SelectObject( hdc, hBrush );
```

When the brush is no longer needed, `DeleteObject` is called to cancel the brush handle.

```
DeleteObject( hBrush );
```

Note that `DeleteObject` is not called for the bitmaps themselves. This is because these bitmaps are resource elements, and they were not created using the `CreateBitmap` function.

Predefined Bitmaps

In addition to loading resource bitmaps, the `LoadBitmap` function can access the Windows predefined bitmaps. For this usage, the `hInst` parameter is specified as `NULL`, and the `lpBitmapName` parameter must be one of the values shown in Table S13.1.

TABLE S13.1: Predefined Windows Bitmaps

Win3.x, Win98/95 or WinNT		Win98/95 or WinNT*	pre-Win3.0**
<code>OBM_CLOSE</code>	<code>OBM_BTNCORNERS</code>		<code>OBM_OLD_CLOSE</code>
<code>OBM_UPARROW</code>	<code>OBM_UPARROWD</code>	<code>OBM_UPARROWI</code>	<code>OBM_OLD_UPARROW</code>
<code>OBM_DNARROW</code>	<code>OBM_DNARROWD</code>	<code>OBM_DNARROWI</code>	<code>OBM_OLD_DNARROW</code>
<code>OBM_RGARROW</code>	<code>OBM_RGARROWD</code>	<code>OBM_RGARROWI</code>	<code>OBM_OLD_RGARROW</code>
<code>OBM_LFARROW</code>	<code>OBM_LFARROWD</code>	<code>OBM_LFARROWI</code>	<code>OBM_OLD_LFARROW</code>
<code>OBM_REDUCE</code>	<code>OBM_REDUCED</code>		<code>OBM_OLD_REDUCE</code>
<code>OBM_ZOOM</code>	<code>OBM_ZOOMD</code>		<code>OBM_OLD_ZOOM</code>
<code>OBM_RESTORE</code>	<code>OBM_RESTORED</code>		<code>OBM_OLD_RESTORE</code>
<code>OBM_MNARROW</code>	<code>OBM_COMBO</code>		
<code>OBM_BTFSIZE</code>	<code>OBM_CHECK</code>		
<code>OBM_SIZE</code>	<code>OBM_CHECKBOXES</code>		

*These four bitmaps are not supported by Windows 3.x.

**All bitmap names with the form `OBM_OLD_XXXX` represent bitmaps used by Windows versions prior to version 3.0.

NOTE

For an application to use any of the `OBM_XXXXXX` constants, the constant `OEMRESOURCE` must be defined before including the `Windows.H` header.

Alternatively, you can use the `MAKEINTRESOURCE` macro to create a `DWORD` value with the bitmap ID as the low-order word and the high-order word `NULL`. Then you can use the resulting value in place of the `lpBitmapName` argument, serving the same purpose.

The predefined bitmaps listed are used in a variety of Windows resources. For example, the `OBM_UPARROW` bitmap should be familiar from Windows 3.x, where it appears in the upper-right corner of every application's frame as the Maximize button. For Windows 2000 (and Windows 9x), the `OBM_ZOOM` bitmap is the current equivalent. The `OBM_CHECK` bitmap, as its name might suggest, is a simple check-mark. The `OBM_SIZE` bitmap provides the diagonal marker that appears in the lower-right corner of a resizable window.

If you would like a simple way to experiment with these, modify the *PenDraw4* demo as follows:

```
case WM_CREATE:
    hBitmap[0] = LoadBitmap( NULL, MAKEINTRESOURCE( OBM_UPARROW ) );
    hBitmap[1] = LoadBitmap( NULL, MAKEINTRESOURCE( OBM_ZOOM ) );
    hBitmap[2] = LoadBitmap( NULL, MAKEINTRESOURCE( OBM_CHECK ) );
    hBitmap[3] = LoadBitmap( NULL, MAKEINTRESOURCE( OBM_SIZE ) );
    break;
```

Remember that the `DeleteObject` function must be called to delete each bitmap handle returned by the `LoadBitmap` function.

NOTE

Because the *PenDraw4* demo uses these bitmap resources as brush patterns, under Windows 98, only the upper-left 8×8 pixels of the bitmap become the brush pattern. If you run the demo under Windows NT/2000, the bitmap patterns will be fully visible.

Old-Style Bitmaps

The old-style bitmaps originated with Windows 1.0. These bitmaps have the principal drawback of being device-dependent, which means that old-style bitmaps are structured to match specific display formats and cannot be conveniently transported to other device contexts.

Functions for Old-Style Bitmaps

Windows provides four principal functions for creating old-style bitmaps:

```
hBitmap = CreateBitmap( cwWidth, cyHeight, nPlanes, nBitsPixel, lpBits );
hBitmap = CreateBitmapIndirect( &bitmap );
hBitmap = CreateCompatibleBitmap( hdc, cxWidth, cyHeight );
hBitmap = CreateDiscardableBitmap( hdc, cxWidth, cyHeight );
```

The `cxWidth` and `cyHeight` arguments define the width and height, in pixels, of the bitmap. As described earlier in the chapter, the `CreateBitmap` function accepts specifications for the number of color planes and number of bits per pixel, matching the image to the device-context requirements.

As alternatives, in the `CreateCompatibleBitmap` and `CreateDiscardableBitmap` functions, the device-context handle (`hdc`) permits Windows to access the number of color planes and the color bits per pixel directly. However, both of these functions create uninitialized bitmap images and require the `SetBitmapBits` function to include image information (see the following discussion of the `SetBitmapBits` and `GetBitmapBits` functions).

The `CreateBitmapIndirect` function uses the structure `BITMAP` to define the bitmap data, including size, colors, and image, in a fashion paralleling the original `CreateBitmap` function. The `BITMAP` structure is defined in `WinGDI.H` as:

```
typedef struct tagBITMAP
{
    LONG    bmType;           // should be 0
    LONG    bmWidth;          // width in pixels
    LONG    bmHeight;         // height in pixels
    LONG    bmWidthBytes;     // width in bytes (must be even)
    WORD    bmPlanes;         // number of color planes
    WORD    bmBitsPixel;      // color bits per pixel
    LPVOID  bmBits;           // pointer to image data
} BITMAP, *PBITMAP, NEAR *NPBITMAP, FAR *LPBITMAP;
```

Bitmap Image Data

The `SetBitmapBits` function is used to copy a char (or byte) array into an existing bitmap, usually an uninitialized bitmap.

```
SetBitmapBits( hBitmap, dwCount, lpBits );
```

As an alternative, the image data can be retrieved from an existing bitmap via the `GetBitmapBits` function.

```
GetBitmapBits( hBitmap, dwCount, lpBits );
```

The `GetBitmapBits` function copies `dwCount` bits from `hBitmap` to the array addressed as `lpBits`. If the size information is not known, you can calculate `dwCount` by first calling the `GetObject` function to retrieve the bitmap structure data:

```
GetObject( hBitmap, sizeof(BITMAP), (LPSTR) &bm );
```

And, once the data is available in `bm`, `dwCount` can be calculated as:

```
dwCount = (DWORD)( bm.bmWidthBytes * bm.bmHeight * bm.bmPlanes );
```

Finally, because these bitmaps are GDI objects, you should use the `DeleteObject` function to cancel the object when it is no longer needed.

```
DeleteObject( hBitmap );
```

Old-Style Monochrome Bitmaps

Earlier in the chapter, we discussed using the `wBricks` array to create an 8×8 monochrome brush from an array of byte values. For bitmaps not intended simply for use with brushes, the 8×8 limitation does not apply, even though each scan line of the bitmap must be an even number of bytes in width (some multiple of 16 bits with zeros used to right-pad the data).

For example, a simple monochrome bitmap consisting of a 9×9 square with two diagonals crossing in the center could be defined as:

1 1 1 1 1 1 1 1 1	1 0 0 0 0 0 0 0 0	= FFh 80h
1 1 0 0 0 0 0 0 1	1 0 0 0 0 0 0 0 0	= C1h 80h
1 0 1 0 0 0 1 0 0	1 0 0 0 0 0 0 0 0	= A2h 80h
1 0 0 1 0 1 0 0 0	1 0 0 0 0 0 0 0 0	= 94h 80h
1 0 0 0 1 0 0 0 0	1 0 0 0 0 0 0 0 0	= 88h 80h
1 0 0 1 0 1 0 0 0	1 0 0 0 0 0 0 0 0	= 94h 80h
1 0 1 0 0 0 1 0 0	1 0 0 0 0 0 0 0 0	= A2h 80h
1 1 0 0 0 0 0 0 1	1 0 0 0 0 0 0 0 0	= C1h 80h
1 1 1 1 1 1 1 1 1	1 0 0 0 0 0 0 0 0	= FFh 80h

To make the bitmap 9×9, each scan line requires 7 pad bits (zeros), for a total width of 16 bits or 2 bytes.

To implement this particular image, the corresponding `BITMAP` structure could be defined as:

```
static BITMAP bm = { 0, 9, 9, 2, 1, 1 };
```

The corresponding image data would be stored in an array of bytes as:

```
static BYTE CheckBox[] =
{ 0xFF, 0x80, 0xC1, 0x80, 0xA2, 0x80, 0x94, 0x80, 0x88,
  0x80, 0x94, 0x80, 0xA2, 0x80, 0xC1, 0x80, 0xFF, 0x80 };
```

NOTE

For old-style bitmaps, the images are coded from the top down. In the new DIB format, images are coded from the bottom up. Of course, because the present example is symmetrical, direction becomes irrelevant.

The simplest method of creating a bitmap from the sample data is to use the `CreateBitmap` function.

```
hBitmap = CreateBitmap( 9, 9, 1, 1, CheckBox );
```

Alternatively, you could use the `CreateBitmapIndirect` function.

```
bm.bmBits = (LPSTR) CheckBox;
hBitmap = CreateBitmapIndirect( &bm );
```

However, there is a potential bug in this format. Because Windows expects to be able to move data around as necessary, the address returned for `CheckBox` may or may not remain valid after it has been assigned. This potential error can be avoided by first creating the bitmap and then transferring the bitmap image to the display (device) context.

```
hBitmap = CreateBitmapIndirect( &bm );
SetBitmapBits( hBitmap, (DWORD) sizeof(CheckBox), CheckBox );
```

Old-Style Color Bitmaps

For color bitmaps, using Windows old-style is both extremely device-dependent as well as quite a bit more complex than for monochrome bitmaps. To illustrate why, following is the 16-color equivalent of `CheckBox`, using only two colors: dark green and white (assuming a standard palette). The bitmap image is calculated as:

```
F F F F F F F F F 0 0 0 = FFh FFh FFh FFh F0h 00h
F F 2 2 2 2 2 F F 0 0 0 = FFh 22h 22h 2Fh F0h 00h
F 2 F 2 2 2 F 2 F 0 0 0 = F2h F2h 22h F2h F0h 00h
F 2 2 F 2 F 2 2 F 0 0 0 = F2h 2Fh 2Fh 22h F0h 00h
F 2 2 2 F 2 2 2 F 0 0 0 = F2h 22h F2h 22h F0h 00h
F 2 2 F 2 F 2 2 F 0 0 0 = F2h 2Fh 2Fh 22h F0h 00h
```



```

F 2 F 2 2 2 F 2 F 0 0 0 = F2h F2h 22h F2h F0h 00h
F F 2 2 2 2 2 F F 0 0 0 = FFh 22h 22h 2Fh F0h 00h
F F F F F F F F 0 0 0 = FFh FFh FFh FFh F0h 00h

```

Again, each scan line is padded to a WORD width by adding three zero (black) pixels at the end of each scan line.

For an EGA/VGA device, this bitmap can be interpreted as a marked checkbox in white against a dark-green background, with each four bits representing the color of one pixel. However, if the display device is, for example, an IBM8514/A, where 8 bits are interpreted as the color value for each pixel, not only will the colors be different, but the image will also be quite different. Or, what about the case where a true-color video is used as the display context and 24 bits of color data are expected for each pixel? The solution is found in the newer device-independent bitmap format described in the next section.

Device-Independent Bitmaps

The device-independent bitmap (DIB) format originally appeared as an extension of the OS/2 Presentation Manager bitmap format (and, perhaps, the only good element to come out of OS/2 version 1.1). This format presents an RGB color table defining all the colors used in the bitmap. Most (if not all) bitmap editors or paint programs automatically create DIB image files. However, because device-independent bitmaps have become so common, the .DIB extension is rarely used; files bearing the .BMP extension are almost always device-independent images, not device-dependent.

The DIB File Format

The DIB image file format consists of several sections: the DIB header, the BITMAP-INFOHEADER, the color table, and the image data. Each of these is described in the following sections.

The DIB File Header

The DIB bitmap file begins with a file header that provides information about the structure of the file itself. The DIB header (defined in `WinGDI.H`) consists of a 14-byte record, shown in Table S13.2.

TABLE S13.2: DIB Header Format

Field	Size	Sample Data	Value	Description
bfType	WORD	42 4D	"BM"	Bitmap ID (constant, all DIBs)
bfSize	DWORD	96 00 00 00	96h	Total file size (example only)
reserved1	WORD	00 00	0h	Set to 0
reserved2	WORD	00 00	0h	Set to 0
bfOffBits	DWORD	76 00 00 00	76h	Offset to bitmap image from first of file (example only)

NOTE

Remember that all data is arranged in **1sb...msb** order. For example, the data bytes 96 00 00 00 represent the value 00000096h, not 96000000h. While this ordering may appear strange, it is simply a firmly entrenched artifact that originated in the early days of computing when the **1sb...msb** (least significant/most significant) order made it faster to process values by storing them in the stack in this fashion. This reverse order made it possible to extract values—one byte at a time as required by 8-bit CPUs—from the stack in the order in which they would be processed.

The BITMAPINFOHEADER Structure

The file header information is followed by a second data header defined by the BITMAPINFOHEADER structure. This data is shown in Table S13.3.

TABLE S13.3: BITMAPINFOHEADER Data

Field	Size	Sample	Value	Description
BiSize	DWORD	28 00 00 00	28h	Size of BITMAPINFOHEADER
BiWidth	LONG	08 00 00 00	8h	Bitmap pixel width
BiHeight	LONG	08 00 00 00	8h	Bitmap pixel height
BiPlanes	WORD	01 00	1h	Color planes (always 1)
BiBitCount	WORD	04 00	4h	Color bits per pixel (1, 4, 8, 24)

Continued on next page

TABLE S13.3 (CONTINUED): BITMAPINFOHEADER Data

Field	Size	Sample	Value	Description
BiCompression	DWORD	00 00 00 00	0h	Compression scheme (0=none)
BiSizeImage	DWORD	20 00 00 00	20h	Bitmap image size (used only if compression is set)
BiXPelsPerMeter	LONG	00 00 00 00	0h	Horizontal resolution (pixels/meter)
BiYPelsPerMeter	LONG	00 00 00 00	0h	Vertical resolution (pixels/meter)
biD1rUsed	DWORD	00 00 00 00	0h	Number of colors used in image
biClrImportant	DWORD	00 00 00 00	0h	Number of important colors (archaic and rarely, if ever, used today)

The BITMAPINFOHEADER contains quite a bit of data about the DIB image. However, as you can see from the example, often several of these fields are left blank, particularly `biXPelsPerMeter` and `biYPelsPerMeter` (the horizontal and vertical resolution). The final two fields, `biD1rUsed` and `biClrImportant` (the number of colors used and the number of important colors), are often used for additional information about custom colors or multiple color palettes; zero values indicate defaults.

Notice also that color is represented only as multiple color bits per pixel, regardless of how a specific device might expect to handle color. Thus, color will be specified as one bit per pixel for monochrome, four for 16-color bitmaps, eight for 256-color bitmaps, or twenty-four for true-color images (16 million colors).

Also, if data compression is used, the data-compression scheme is identified together with the actual size of the uncompressed bitmap (in bytes), thus providing a redundancy check for use in decompressing the image. Four compression schemes are defined, as shown in Table S13.4.

TABLE S13.4: Compression Formats and Identifiers

Constant	Value	Comment
BI_RGB	0	No compression used
BI_RLE8	1	8-bit run-length encoding format
BI_RLE4	2	4-bit run-length encoding format
BI_TOPDOWN	4	

NOTE

Despite provisions for identifying compression formats, many bitmap editors (or paint programs) do not support (or recognize) compressed image data.

The DIB BITMAP Color Table

The DIB color table follows the BITMAPINFOHEADER. This table consists of a series of RGBQUAD structures. These are read, in order, with the first byte blue, the second green, the third red, and the fourth byte in each quad set to zero.

The `biBitCount` field identifies the number of RGBQUAD structures. For a monochrome image, this field is set as 1 color bit. Two RGBQUAD records are required to identify the foreground and background colors. If `biBitCount` is 4, 16 RGBQUAD color identifiers are needed. If `biBitCount` is 8, 256 RGBQUAD values are required.

If the `biClrUsed` field is nonzero, this value (instead of the `biBitCount` field) identifies the number of RGBQUAD structures in the color table.

Table S13.5 shows the default color values for a VGA 16-color palette expressed as RGBQUAD values.

TABLE S13.5: A Sample Color Palette for a DIB Bitmap

Palette Entry	RGBQUAD Data	Color Value			Approximate Color
		R	G	B	
0	00 00 00 00	00	00	00	Black
1	00 00 80 00	80	00	00	Dark Red
2	00 80 00 00	00	80	00	Dark Green
3	00 80 80 00	80	80	00	Gold Green
4	80 00 00 00	00	00	80	Dark Blue
5	80 00 80 00	80	00	80	Purple
6	80 80 00 00	00	80	80	Blue Gray
7	80 80 80 00	80	80	80	Dark Gray
8	C0 C0 C0 00	C0	C0	C0	Light Gray

Continued on next page

TABLE S13.5 (CONTINUED): A Sample Color Palette for a DIB Bitmap

Palette Entry	RGBQUAD Data	Color Value			Approximate Color
		R	G	B	
9	00 00 FF 00	FF	00	00	Light Red
10	00 FF 00 00	00	FF	00	Light Green
11	00 FF FF 00	FF	FF	00	Yellow
12	FF 00 00 00	FF	00	00	Light Blue
13	FF 00 FF 00	FF	00	FF	Magenta
14	FF FF 00 00	00	FF	FF	Cyan
15	FF FF FF 00	FF	FF	FF	White

The DIB BITMAP Image

The final section of the bitmap file is the bitmap image itself. The arrangement of this section partly depends on the number of colors (as reported by the `biBitCount` field), but it is also affected by two other factors, which are constant for all bitmaps:

- Each row of the bitmap image must be a multiple of four bytes (a `DWORD` multiple). Each data row begins with the leftmost pixel of the scan line and is right-padded with zeros, as necessary.
- Unlike the original bitmap format (Windows 1.0 or 2.0), the bitmap format for DIBs begins with the bottom scan line in the image, not the top.

For a monochrome bitmap—one color bit per pixel—the bit image begins with the most-significant bit of the first byte in each row. If the bit value is zero (0), the first RGBQUAD color value is used (background). If the bit value is one (1), the second RGBQUAD value is used (foreground).

For a monochrome bitmap, the BRICKS bitmap data would be coded as:

80 80 80 FF 08 08 08 FF

This data would break down, as a pixel image, as:

```

1 1 1 1 1 1 1 1 // FFh
0 0 0 0 1 0 0 0 // 08h
0 0 0 0 1 0 0 0 // 08h
0 0 0 0 1 0 0 0 // 08h
1 1 1 1 1 1 1 1 // FFh
1 0 0 0 0 0 0 0 // 80h
1 0 0 0 0 0 0 0 // 80h
1 0 0 0 0 0 0 0 // 80h

```

Again, as a reminder, notice that the image data, from left to right, appears in the image from bottom to top, not top down.

For a 16-color bitmap, as used in the Bricks.BMP file with four bits per pixel, each pixel is represented by a four-bit value that serves as an index to the palette entries in the table (as shown in Table S13.5). The color bitmap image appears as:

```

81 11 11 11 81 11 11 11 81 11 11 11 88 88 88 88
11 11 81 11 11 11 81 11 11 11 81 11 88 88 88 88

```

The color image data is decoded as:

```

8 8 8 8 8 8 8 8 // 88h 88h 88h 88h
1 1 1 1 8 1 1 1 // 11h 11h 81h 11h
1 1 1 1 8 1 1 1 // 11h 11h 81h 11h
1 1 1 1 8 1 1 1 // 11h 11h 81h 11h
8 8 8 8 8 8 8 8 // 88h 88h 88h 88h
8 1 1 1 1 1 1 1 // 81h 11h 11h 11h
8 1 1 1 1 1 1 1 // 81h 11h 11h 11h
8 1 1 1 1 1 1 1 // 81h 11h 11h 11h

```

In a similar fashion, for a 256-color bitmap, each pixel is represented by a byte value indexing the 256 entries in the color table.

For a 24-bit-per-pixel color bitmap, with the `biClrUsed` field specified as zero, instead of a color table with 16 million entries (predicating a minimum file size of 64MB just for the color table), no color table is used. Each pixel is represented by a three-byte `RGBColor` value. If `biClrUsed` is not zero, a color table is included and pixels are indexed to the table.

OS/2 Bitmaps

OS/2 version 1.1 and later uses a bitmap structure that is very similar to Windows, with only two principal structure changes. First, instead of a `BITMAPINFOHEADER` structure, OS/2 uses a `BITMAPCOREHEADER` structure, which is defined in `WinGDI.H` as:

```
typedef struct tagBITMAPCOREHEADER
{
    DWORD   bcSize;           // offset to color table
    WORD     bcWidth;
    WORD     bcHeight;
    WORD     bcPlanes;
    WORD     bcBitCount;
} BITMAPCOREHEADER, FAR *LPBITMAPCOREHEADER,
  *PBITMAPCOREHEADER;
```

Second, instead of a color table consisting of `RGBQUAD` records, the OS/2 bitmaps use `RGBTRIPLE` records.

Perhaps the simplest method of identifying the two formats is to check the two byte values in the image file for the value `BM`, identifying Windows bitmap format. If these two bytes do not identify Windows format, the OS/2 structure can be confirmed by testing the first `DWORD` value in `BITMAPIMAGEHEADER`/`BITMAPCOREHEADER` structures to determine the structure size.

Bitmap Dimension Functions

Windows supplies two bitmap dimension functions: `SetBitmapDimensionEx` and `GetBitmapDimensionEx`. However, despite what the names might initially suggest, these two functions do not deal with the pixel dimensions of a bitmap because, once an image is created, the pixel size of the image cannot be changed. Instead, this function pair provides a means of setting or retrieving bitmap dimensions in logical units (the `MM_LOMETRIC` mode is assumed). These dimensions are not used by the GDI for screen display but may be used by other applications to scale bitmaps that have been exchanged using the clipboard, DDE, or other channels.

The `SetBitmapDimensionEx` and `GetBitmapDimensionEx` functions are called as:

```
SetBitmapDimensionEx( hBitmap, xUnits, yUnits, lpSize );
GetBitmapDimensionEx( hBitmap, lpSize );
```

The `lpSize` variable returns with the previous size data (when new dimensions are set) or the current size data (when the `get` function is called). The `SIZE` data structure is defined in `WinDef.H` as:

```
typedef struct tagSIZE
{   LONG   cx;
    LONG   cy; } SIZE, *PSIZE, *LPSIZE;
```

NOTE

In general, the two bitmap size fields (`biXPelsPerMeter` and `biYPelsPerMeter`) are set to zero except when needed by special circumstances, such as when you are providing additional rendering (sizing) information for hard-copy devices. These two values are rarely employed and may be overridden (or ignored) even when set.

Device-Independent Bitmap Creation

Ideally, it would be nice if Windows supplied a simple function to create (or load) and display a bitmap, requiring only a device context, bitmap name, and position. This function might look something like this:

```
DrawBitmap( hwnd, lpBitmapName, xPos, yPos );
```

However, even though bitmaps are both important and integral to Windows, no such basic display function is provided. Instead, Windows provides a series of bitmap primitives that can be used to construct a number of the missing high-level bitmap handlers, beginning with a function titled, appropriately, `DrawBitmap`.

The following sections describe the basic steps required to create and display a device-independent bitmap.

Step One: Providing a Global Instance Handle

Up to this point, all the program examples have included one provision which, thus far, has not been used, needed, explained, or (most likely) even noticed. The provision in reference, which does have more than a few uses, begins with the global handle declaration:

```
HANDLE hInst;
```


In the `WinMain` procedure, the `hInst` variable is assigned as:

```
hInst = hInstance;
```

Without this provision in the *PenDraw4* demo, for example, the `LoadBitmap` instructions in response to the `WM_CREATE` message in `WndProc` would need to have been executed in the `WinMain` procedure using the `hInstance` handle.

Although there are other ways to retrieve an application's instance handle, the global instance handle costs a mere 16 bits of overhead memory, so why bother with false economies?

Once the global `hInst` instance handle is available, the `LoadBitmap` function can be implemented within our theoretical `DrawBitmap` function without invoking special provisions to retrieve the application's instance handle.

Step Two: Defining `DrawBitmap`

The basic form of `DrawBitmap` is called with four parameters: the window handle (`hwnd`), the bitmap name (`lpName`), and `x` and `y` coordinates to position the bitmap. And, as a result, `DrawBitmap` displays a bitmap at the coordinates specified. Ergo, the function declaration begins as:

```
BOOL DrawBitmap( HWND hwnd, LPSTR lpName,  
                 int xPos, int yPos )  
{
```

NOTE

`DrawBitmap` is also provided with the capability to return a Boolean result, reporting success or failure. But as with most C functions, the returned value may be used or ignored, as desired.

A few local variables will be needed, and they are declared as:

```
HDC      hdc, hdcMem;  
BITMAP   bm;  
HBITMAP  hBitmap;
```

Declarations finished, the function is now ready to load a bitmap from the resource segment of the .EXE program. Notice, however, that this is also the point where the global `hInst` handle becomes essential.

```
if( !( hBitmap = LoadBitmap( hInst, lpName ) ) )  
    return( FALSE );
```

Of course, if the load operation fails, `DrawBitmap` will immediately terminate, returning `FALSE`. This is the only error check provided.

If successful, once the bitmap is loaded, the next step is to establish a suitable device context to display the bitmap.

Step Three: Creating the Device Context

Unlike in DOS, where once a graphics mode has been established anything can be written (drawn) on the screen, under Windows, a bitmap image cannot be drawn (or copied) directly to the display-device context. Instead, before the bitmap image can be drawn, a separate device context is created. This is created as a memory device context (with no immediate connection to an output device), using the `hdcMem` variable declared local to the `DisplayBitmap` function.

However, the application's actual output device context cannot simply be ignored. Therefore, the next order of business is to retrieve a handle to the application's device context.

```
hdc = GetDC( hwnd );  
hdcMem = CreateCompatibleDC( hdc );
```

The trick here is that a reference device context (`hdc`) is needed before the `CreateCompatibleDC` function can be called to create the memory context (`hdcMem`). The memory device context is simply a block of memory that acts as an analog for the real display context. For a bitmap, the memory device context can be used to prepare an image in memory before transferring the image to the display context (to the screen or another output device).

When the memory device context is created, the GDI automatically assigns a "display surface" sized for a 1×1 monochrome image; that is, a one-pixel monochrome bitmap. But, while this is hardly sufficient space for any real operations, this deficiency can be corrected immediately by calling `SelectObject` to use the bitmap that was loaded a moment before as the active object for the device context:

```
SelectObject( hdcMem, hBitmap );  
SetMapMode( hdcMem, GetMapMode( hdc ) );
```

After selecting the bitmap into the memory context, `SetMapMode` assigns the mapping mode used by the active device context (`hdc`) to the memory device context (`hdcMem`), thus making the memory image of the bitmap a suitable match for the output device.

At this point, the bitmap has become the active object for the memory device context, while the memory device has the same mapping mode as the actual

device context. But the job isn't done yet; there is still quite a bit of information that needs to be transferred from the source bitmap (hBitmap) to the local bitmap record (bm).

Step Four: Transferring Bitmap Definition Data

The `GetObject` function can be used to transfer most of the information needed to fill the buffer (bm) to define the logical object (the selected bitmap). For a bitmap, `GetObject` returns the width, height, and color format information. This function is called as:

```
GetObject( hBitmap, sizeof( BITMAP ), (LPSTR) &bm );
```

But still, the actual image data has not been retrieved yet. This operation comes next.

Step Five: Retrieving Image Data

The `BitBlt` (short for bit-block-transfer and pronounced "bit-blit"), `PutBlt`, and `StretchBlt` functions compose Windows pixel-manipulation power operations. However, while each of these function names implies a block-transfer operation, there's more involved here than simply copying bits from one memory location to another. Instead, there is also a choice of raster operations, as will be explained in a moment.

While not the simplest of the three operations, the `BitBlt` operation is, for the present purpose, the operation of choice. It is used to complete the task of writing the bitmap image to the client window:

```
BitBlt( hdc, xPos, yPos, bm.bmWidth, bm.bmHeight,  
        hdcMem, 0, 0, SRCCOPY );
```

The `BitBlt` operation moves the bitmap image from the source device (hdcMem) to the destination device (hdc), with the `xSrc` and `ySrc` parameters (0,0 in the example) specifying the origin (in the source device context) of the bitmap to be transferred.

The `xPos`, `yPos`, `bm.bmWidth`, and `bm.bmHeight` parameters provide the origin and rectangle size (in the destination device context) to be filled by the bitmap image. Unlike many previous operations, instead of `RECT` rectangular coordinates, the origin point is specified in device-context coordinates. The width and height are passed as logical units, not as device coordinates. As demonstrated, these last two values are taken directly from the bitmap data. Optionally, you can assign the width and height values on some other basis.

The final parameter is a ternary raster-operation code specifying how the GDI will combine colors between a current brush (pattern), the source image, and any existing destination image. For the `DrawBitmap` operation, the `SRCCOPY` ROP copies the source bitmap image directly to the destination (hdc).

The 15 principal ternary raster operations are defined in `WinGDI.H` and listed in Table S13.6.

TABLE S13.6: Raster Operation Codes (Ternary Raster Ops)

Constant	Operation	Description
<code>SRCCOPY</code>	<code>Dest = Source</code>	Source copied to destination
<code>SRCPAINT</code>	<code>Dest = Source Dest</code>	Destination ORed with source
<code>SRCAND</code>	<code>Dest = Source & Dest</code>	Source ANDed with destination
<code>SRCINVERT</code>	<code>Dest = Source ^ Dest</code>	Source XORed with destination
<code>SRCERASE</code>	<code>Dest = Source & !Dest</code>	Destination inverted before ANDing with source
<code>NOTSRCCOPY</code>	<code>Dest = !Source</code>	Inverted source copied to destination
<code>NOTSRCERASE</code>	<code>Dest = !Source & !Dest</code>	Inverted destination ANDed with inverted source
<code>MERGECOPY</code>	<code>Dest = Source & Patt</code>	Source ANDed with pattern
<code>MERGEPAINT</code>	<code>Dest = !Source Dest</code>	Destination ORed with inverted source
<code>PATCOPY</code>	<code>Dest = Patt</code>	Pattern copied to destination
<code>PATPAINT</code>	<code>Dest = Patt !Source Dest</code>	Pattern ORed with inverted source, result ORed with destination
<code>PATINVERT</code>	<code>Dest = Patt ^ Dest</code>	Pattern XORed with destination
<code>DSTINVERT</code>	<code>Dest = !Dest</code>	Destination inverted
<code>BLACKNESS</code>	<code>Dest = Black (0)</code>	Destination turned black
<code>WHITENESS</code>	<code>Dest = White (1)</code>	Destination turned white

Raster operations involving monochrome images are fairly straightforward: Bits will be either on or off according to the logical operations selected. For color bitmaps, however, the GDI executes separate operations for each color plane or for each set of color bits, depending on the device-context organization. The best

way to understand these operations is to experiment, preferably with relatively simple bitmaps and patterns.

Step 6: Cleaning Up

Calling the `BitBlt` API completes the task of drawing the bitmap, but before `DrawBitmap` returns, some cleanup is still required. This is accomplished as:

```
ReleaseDC( hwnd, hdc );  
DeleteDC( hdcMem );  
DeleteObject( hBitmap );  
return( TRUE );
```

Initially, three local memory allocations were made, returning three handles as `hdc`, `hdcMem`, and `hBitmap`. The first of these is simply released rather than being deleted; that is, the `hdc` handle is released, but the application device context is not deleted. The local memory device context, however, is deleted entirely, deallocating all memory involved, not just the memory handle. The locally allocated and loaded bitmap is treated in a similar fashion.

After this cleanup is completed, the `DrawBitmap` function is free to return, reporting success.

The `DrawBitmap` function is demonstrated in the *PenDraw5* demo, which is discussed after we cover one more bitmap operation.

Stretching Bitmaps

Drawing a bitmap using a one-for-one transfer is probably the most common operation. However, another bitmap operation you may find useful is provided by `StretchBlt`, which permits stretching or distorting a bitmap to fit any rectangular space desired. This function moves a bitmap from a source rectangle to a destination rectangle, stretching or compressing the bitmap as appropriate to fit the destination dimensions.

Calling the `StretchBlt` operation is similar to calling `BitBlt`, but with two differences:

```
BitBlt( hdc,      xPos, yPos, xWidth,  yHeight,  
        hdcMem, xOrg, yOrg, dwRasterOp );  
StretchBlt( hdc,      xPos, yPos, xWidth,  yHeight,  
            hdcMem, xOrg, yOrg, xWidthOut, yHeightOut, dwRasterOp );
```

The `StretchBlt` operation is called with two additional parameters specifying the destination width and height; for the `BitBlt` operation, source and destination width and height are the same. It is precisely this difference that instructs `StretchBlt` to stretch or compress the bitmap during transfer. Since `xWidth/xWidthOut` and `yWidth/yWidthOut` are independent, the bitmap could be stretched along one axis and compressed along another.

As with the `BitBlt` operation, the `dwRasterOp` specification controls how the source and destination (if any) bitmaps are combined during the `StretchBlt` operation.

`StretchBlt` operations are not necessarily limited to resizing images. You can also use `StretchBlt` to create a mirror image of a bitmap (laterally or vertically), by specifying different the signs for the source and destination width or the source and destination height. For example, if the destination width is negative and the source width is positive, `StretchBlt` creates a mirror image rotated about the vertical axis (swapping left for right). Likewise, for a difference in sign of the height parameters, the image is mirrored along the horizontal axis. If both pairs are opposite in sign, the image is simply rotated 180° but without mirror inversion.

Because the `StretchBlt` operation resizes a bitmap image, one additional factor controls how data is added or subtracted to create the new image: the `StretchBlt` mode. You set the active mode by calling the `SetStretchBltMode` function as:

```
SetStretchBltMode( hdc, nStretchMode );
```

Four `StretchBlt` modes are defined in `WinGDI.H`, as described in Table S13.7.

TABLE S13.7: `StretchBlt` Modes

Constant	Value	Description
<code>BLACKONWHITE</code>	1	Eliminated lines are ANDed with retained lines; preserves black pixels at expense of white
<code>WHITEONBLACK</code>	2	Eliminated lines are ORed with retained lines; preserves white pixels at expense of black
<code>COLORONCOLOR</code>	3	Eliminated lines are deleted without preserving information
<code>HALFTONE</code>	4	Color information in destination approximates source pixels, averaging information from source to destination

The BLACKONWHITE and WHITEONBLACK modes are typically used to preserve the background or foreground pixels in monochrome bitmaps, respectively. The COLORONCOLOR and HALFTONE modes are typically used to preserve color in color bitmaps, with the principal difference between the two being that the HALFTONE mode produces higher-image quality but does so at the expense of execution time.

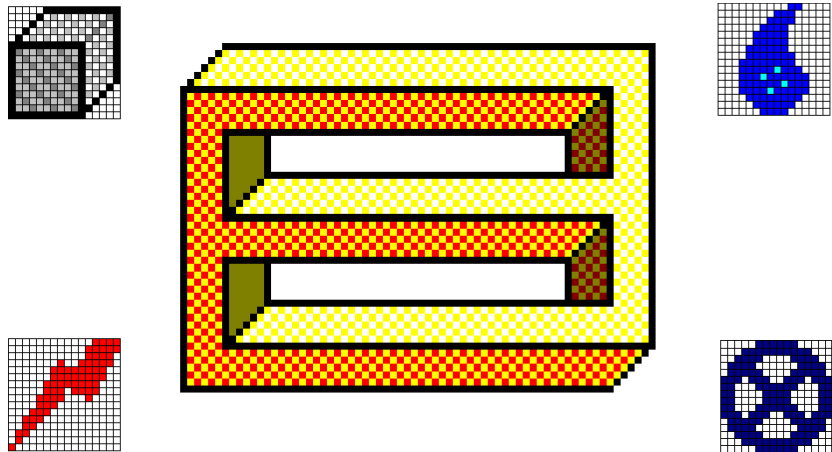
The *PenDraw5* Demo: Displaying Device-Independent Bitmaps



The *PenDraw5* demo demonstrates the `DrawBitmap` function described earlier, as well as the `BitBlt` and `StretchBlt` API functions. *PenDraw5* requires five bitmaps, four 16×16 images, and one 40×70 image, as illustrated in Figure S13.5.

FIGURE S13.5:

Five bitmap images used in the *PenDraw5* demo



TIP

You can use bitmaps other than the ones shown in Figure S13.5 if you prefer. Just be sure to make the appropriate changes in the source and resource codes to identify the desired bitmaps.

Initially, the `DrawBitmap` function draws all five bitmaps, placing the four smaller bitmaps at the corners of the client window and the larger bitmap in the

center. For each of these, the bitmap is drawn with the upper-left corner of the image at the coordinates specified. Several variations are also used in the demo:

- `DrawCenBitmap` centers each image (horizontally and vertically) on the coordinate points.
- `LineGraph` uses a brief array of data to position a series of smaller bitmaps in a form appropriate for a simple line graph.
- `StretchBitmap` uses the `StretchBlt` API to resize a bitmap to fit a specified rectangle.
- `StretchBitmap2Client` stretches a bitmap to fill the entire client window.
- `MoveBitmap` tracks mouse movement by repositioning a bitmap each time the left mouse button is pressed or, if the button is held down, by tracking the mouse cursor directly. (Bitmaps are not well-suited to this last operational format, `MoveBitmap`; this is intended more as a demonstration than as a serious example of practical programming.)

NOTE

The *PenDraw5* demo is included on the CD that accompanies this book, in the Supplement 13 folder.

This chapter described two programs that demonstrate bitmap operations: The *PenDraw4* demo shows how to use resource bitmaps to create brushes, and the *PenDraw5* demo demonstrates how to work with device-independent bitmaps. Another demo, *SVGA_Win*, is also included on the CD to demonstrate several types of color palettes.

Bitmap operations are very powerful tools with a wide variety of uses, extending well beyond the few examples employed in this chapter. For example, bitmaps can be copied from the screen itself, generated or modified off-screen, saved as external files, or cut and pasted from one window to another.

Typefaces and Styles

- Text-output features
- Windows' default fonts
- Logical font selection
- Font characteristics
- Font sizing and mapping modes

Windows 2000 provides a selection of typefaces and the capabilities to vary these typefaces with considerable convenience and flexibility. You've seen examples of text operations in previous chapters. Now it's time to examine a few of the advanced text features, including font selection, font sizing, text justification, character weighting, and other stylistic changes.

Before we get to fonts and typefaces, we'll take a closer look at some of Windows' text-output features. There are a few that we haven't covered yet, and they deserve some introduction or further explanation.

Text-Output Features

Thus far in the book, most text-display examples have used one of these two general formats:

```
TextOut( hdc, xPos, yPos, lpStr, nCount );
```

```
TextOut( hdc, xPos, yPos, szBuff, wsprintf( szBuff, ... ) );
```

The second format employs the `wsprintf` function both to return the character count required by the `TextOut` API function and to create a formatted string. But, regardless of the format used, previous examples have, almost exclusively, used the default, flush-left text alignment.

Using the MFC classes, a third format for text display has appeared as:

```
pDC->TextOut( xPos, yPos, csBuff );
```

In this format, no handle to the device context is passed because the `TextOut` function is a member of the `CDC` class. Likewise, because the `CString` instance contains the string-length information, this value is also not required as an argument.

All of these text formats provide only the simplest form of text display, without alignment, font selection, or special formatting.

Text Alignment

The `SetTextAlign` function provides control not only over the horizontal text alignment, relative to the specified x- and y-coordinate specifications, but also over vertical alignment and current position updating. `SetTextAlign` is called as:

```
SetTextAlign( hdc, wFlags );
```

The `SetTextAlign` settings affect text displayed using both the `TextOut` and `ExtTextOut` functions. The `wFlags` argument consists of one or more text-alignment specifications combined using the OR operator. Eight alignment constants are defined in `WinGDI.H`. These are listed in Table S14.1.

NOTE

The bounding rectangle is a rectangle surrounding the text string, passed as an argument to the `TextOut` or `ExtTextOut` functions.

TABLE S14.1: Horizontal Text Alignment Flags

Flag ID	Bit Flags	Value	Comments
Vertical Alignment at yPos			
TA_TOP *	0000 0000	0	Aligns with top of bounding rectangle
TA_BASELINE	0001 1000	24	Aligns with baseline of selected font
TA_BOTTOM	0000 1000	8	Aligns with bottom of bounding rectangle
Horizontal Alignment at xPos			
TA_LEFT *	0000 0000	0	Aligns with left side of bounding rectangle
TA_RIGHT	0000 0010	2	Aligns with right side of bounding rectangle
TA_CENTER	0000 0110	6	Aligns with horizontal center of bounding rectangle (current position is not affected)
Current Position Control			
TA_NOUPDATECP *	0000 0000	0	Does not update current position after <code>TextOut</code> or <code>ExtTextOut</code> calls
TA_UPDATECP	0000 0001	1	Updates current position after <code>TextOut</code> or <code>ExtTextOut</code> calls
Combined Flags			
TA_MASK	0001 1111	31	TA_BASELINE + TA_CENTER + TA_UPDATECP

* The default flags are TA_LEFT, TA_TOP, and TA_NOUPDATECP

Because not all fonts are written horizontally (for example, the Japanese Kanji font is written vertically), two additional flag values substitute for the `TA_BASELINE` and `TA_CENTER` flags. These are defined as shown in Table S14.2.

TABLE S14.2: Vertical Text Alignment Flags

Constant	Replaces	Comments
<code>VTA_BASELINE</code>	<code>TA_BASELINE</code>	Aligns reference point with baseline of text
<code>VTA_CENTER</code>	<code>TA_CENTER</code>	Aligns reference point vertically with center of bounding rectangle

The `SetTextAlign` function returns an unsigned integer specifying the previous text alignment, or if an error occurs, `ERROR` is returned.

Extended Text Output Options

The `ExtTextOut` function expands on the `TextOut` function. It adds a rectangle specification that can be used for clipping, opaquing, or both, as well as a pointer to an array of data to control character spacing. The `ExtTextOut` function is called as:

```
ExtTextOut( hdc, xPos, yPos, fOptions, lpRect,
            szString, nCount, lpDx );
```

The `hdc`, `xPos`, `yPos`, `szString`, and `nCount` parameters perform in precisely the same fashion as with the `TextOut` function. The differences are found in the `fOptions`, `lpRect`, and `lpDx` parameters.

fOptions This parameter may be `NULL`, or it may be either or both (ORed) of the following flag values:

- `ETO_CLIPPED`, which clips the text to fit the rectangle specification
- `ETO_OPAQUE`, which fills the rectangle using the current background color

lpRect This parameter points to a `RECT` structure specifying the enclosing rectangle, or it may be passed as `NULL`.

lpDx This parameter points to an array of integer values that specify the distance between adjacent character cells in logical units. For example, element `lpDx[i]` sets the spacing between the origins of the characters `szString[i]` and `szString[i+1]`. If `lpDx` is `NULL`, the default character spacing is used.

By default, the current position is not updated by calls to `ExtTextOut`. However, if the `SetTextAlign` function is called to set `TA_UPDATECP`, two changes occur. First, the initial call to `ExtTextOut` uses the `xPos`, `yPos` parameters, updating the current position after drawing the text parameter. Then, on the second and subsequent calls to `ExtTextOut`, the `xPos` and `yPos` parameters will be ignored and only the current position data will be used. The current position will continue to be updated with the results of each call.

Tabbed Text

Conventionally, graphics text functions have not included any tab provisions, an oversight which is now corrected by the `TabbedTextOut` function. This function permits an output string to be tabbed according to spacing arguments specified in an `lpnTabStopPositions` array. The `TabbedTextOut` function is called as:

```
TabbedTextOut( hdc, xPos, yPos, szString, nCount,  
              nTabPositions, lpnTabStopPositions, nTabOrigin );
```

The first five parameters are the same as the equivalent parameters in the `TextOut` function. The difference is that tab characters can be included in the `szString` parameter as embedded `\t` (or `0x09`) characters.

The other parameters are as follows:

nTabPositions This parameter is an integer argument specifying the number of tab stops to be set (the number of entries in the `lpnTabStopPositions` array) or the number of tab stops to be used. Three variations may be used:

- If `nTabPositions` is zero (0) and `lpnTabStopPositions` is `NULL`, all tabs are expanded to eight times the average character width.
- If `nTabPositions` is one (1), all tabs are incremented by the first distance specified in the `lpnTabStopPositions` array.
- If `lpnTabStopPositions` contains multiple values, subsequent tabs are set according to these values up to the number specified by `nTabPositions`.

lpnTabStopPositions This parameter points to an array of tab stops (in increasing order) defined in device units, or it may be `NULL`.

nTabOrigin This parameter is an integer specification, in device units, specifying an initial offset from which the tab specifications are expanded. The **nTabOrigin** argument also allows an application to call **TabbedTextOut** two or more times for a single line, specifying a new offset each time.

Gray Text

The **GrayString** function draws text using a gray brush. It draws gray text by first writing the text in a memory context as a bitmap, graying the bitmap, and then copying the bitmap to the text display. The drawn text is grayed independently of any brush or background color active in the device context used for the display. The font used is the font currently selected in the device context specified by the **hdc** parameter.

The **GrayString** function is called as:

```
GrayString( hdc, hBrush, lpOutputFunc, lpData, xPos, yPos, nWidth, nHeight );
```

The parameters are as follows:

hdc This parameter specifies the device context where the grayed string will be displayed.

hBrush This parameter identifies the brush to be used to gray the text.

lpOutputFunc This parameter is an optional procedure instance address for an application-supplied function to be used to draw the string. If this is specified as **NULL**, the **TextOut** function will be used.

lpData This parameter may be a pointer to data to be passed to the **lpOutputFunc** function, or if **lpOutputFunc** is **NULL**, must be a pointer to the string to be displayed.

xPos/yPos These parameters specify, in device coordinates, the starting position of a rectangle bounding the string displayed.

nWidth/nHeight These parameters specify the width and height, in device units, for the rectangle enclosing the text display. If either parameter is zero (0) and **lpData** is a pointer to a string, **GrayString** calculates the width or height.

A **FALSE** result is returned if the **GrayString** function fails, if the **lpOutputFunc** returns failure, or if memory limitations prevent the bitmap from being created.

TIP

You can also draw grayed strings on any device that supports a solid-gray color, without using the `GrayString` function, by using the system color `COLOR_GRAYTEXT`. To do this, call `GetSysColor` to retrieve the color value for `COLOR_GRAYTEXT`. If the result is not zero (0), call `SetTextColor` to select this color before drawing the string directly. If the returned color value is zero, grayed text can only be drawn using the `GrayString` function.

Multiple Text Lines

The `DrawText` function displays formatted text within a specified rectangular area. Unlike the other functions, `DrawText` is specifically designed to display multiple lines, inserting line breaks as required to format the text within the indicated rectangle. The `DrawText` function is called as:

```
DrawText( hdc, szString, nCount, lpRect, wFormat );
```

The `hdc`, `szString` and `nCount` parameters are used to identify the device context, the string to be printed, and the number of characters in the string. The `lpRect` argument is a pointer to a `RECT` structure identifying a rectangle, in device coordinates, where the text will be drawn.

The fifth argument, `wFormat`, is an unsigned integer and consists of an ORed combination of the flags listed in Table S14.3.

TABLE S14.3: DrawText Format Flags

Constant	Comments
Horizontal Justification	
DT_LEFT	Aligns text flush-left
DT_CENTER	Centers text
DT_RIGHT	Aligns text flush-right
Vertical Justification	
DT_TOP	Top-justifies text (single line only)
DT_VCENTER	Centers text vertically (single line only)
DT_BOTTOM	Bottom-justifies text; must be combined with DT_SINGLELINE

Continued on next page

TABLE S14.3 (CONTINUED): DrawText Format Flags

Constant	Comments
Format and Spacing Instructions	
DT_EXTERNALLEADING*	Adds font external leading to line spacing
DT_NOCLIP*	Disables clipping to rectangle (operation is marginally faster)
DT_SINGLELINE	Sets single line only; carriage returns and line feeds do not produce line breaks
DT_WORDBREAK	Enables automatic line breaks at word boundaries as required to fit text to rectangle
DT_EXPANDTABS	Expands tab characters (default is 8 times average character width per tab)
DT_TABSTOP	Sets tab stops using bits 15-8 of the high byte of the low word in wFormat to specify the number of characters for each tab (if zeros, default spacing is used)
DT_NOPREFIX*	Turns off processing of prefix character
DT_INTERNAL *	Not documented
Automatic Rectangle Calculation	
DT_CALCRECT*	Enables automatic calculation of rectangle area but does not draw actual text

*Cannot be used with the DT_TABSTOP flag.

Here are a few additional notes about two of the flags listed in Table S14.3:

- The DT_NOPREFIX flag disables the use of the ampersand (&) character to underline the character immediately following. When DT_NOPREFIX is not set (characters following the ampersand will be underlined), an ampersand can be entered as &&, producing a single & as output.
- The DT_CALCRECT flag enables automatic calculation of the rectangle area. If there are multiple lines of text, DrawText uses the rectangle width specified by the lpRect parameter, extending the base of the rectangle to bound the last line of text. If there is only one line of text, DrawText modifies the width (right size) to bound the last character in the line. In both cases, DrawText returns the height of the formatted text but does not draw the actual text.

Device-Context Elements

Along with the text-output functions and their flags, the text display is also governed by the active device context. Elements specified by the device context include not only the foreground and background colors, but also how the text-display pixels are combined with the existing background image.

By default, when text is drawn, the text background (the area between and around characters) is also filled in using the background color. This drawing mode is the OPAQUE background mode, but it can be changed by calling the `SetBkMode` function.

```
SetBkMode( hdc, nMode )    // OPAQUE or TRANSPARENT
```

The foreground and background color functions have been used in previous examples in this book. In general, they are called as:

```
SetTextColor( hdc, rgbColor );  
SetBkColor( hdc, rgbColor );
```

As with pen and brush colors, the `rgbColor` argument is converted to the nearest solid color supported by the active device. Dithered colors, which are permitted with brushes, are not supported for text or pen displays.

Rather than wondering what colors might be supported, however, the two preceding API calls can be rewritten to request colors that are known to be supported:

```
SetTextColor( hdc, GetSysColor( COLOR_WINDOWTEXT ));  
SetBkColor( hdc, GetSysColor( COLOR_WINDOW ));
```

The default colors for the foreground and background are, respectively, black and white. If you want to change these colors, it is also useful to include a provision (in `WndProc`) to repaint the entire client window when the changes occur. Since no system color changes can be made without issuing a notification message to all applications, the simplest method is to include a `WM_SYSCOLORCHANGE` response:

```
case WM_SYSCOLORCHANGE:  
    InvalidateRect( hwnd );  
    break;
```

Fonts and Typefaces

An obvious prerequisite for a text display is one or more fonts with which to create the display. Under DOS (in conventional text mode), the system hardware—generally, the video card itself—supplied the display font in the form of a ROM-based, bitmapped character set tailored to the device’s display capabilities. Thus, CGA video cards supplied one set of bitmaps, EGA video another, and VGA still a third.

Of course, all of this was quite transparent to the software. Applications had no need to ask or to know what the display characteristics consisted of, or even the display’s capabilities. Applications simply wrote to the output, as ASCII character codes, and let the hardware take care of the rest. Earlier displays were limited to a single typeface and, essentially, a single type size.

Today, in a graphics environment, the old-style text displays are gone. Graphics displays can not only mix text, graphics, and colors, but they can use many different character fonts (typefaces), in many different sizes. Furthermore, they can vary typeface and size in a variety of styles, such as bold and italic, and in many cases, may also vary font widths, slants, and weights.

Programmers now have a wide range of flexibility in handling of graphics text displays. But, to make use of these opportunities, it may help to understand both the origins of type fonts and the characteristics which determine fonts.

Reminiscences of a Printer’s Devil

In personal terms (primarily because of a long personal history in the newspaper business beginning long before electronic typesetting), the word *typeface* conjures images of large flat trays of small compartments filled with individual metal characters in an assortment of sizes and typeface designs.

Most of the typesetting, of course, was accomplished by a huge and intricate machine known as a linotype, operated by a highly skilled (and very well paid) individual who knew both the precision mechanics of the triple keyboard, as well as the massive armatures and injection molds that for many decades produced newspapers, books, and the bulk of all manner of printed material.

Continued on next page

Larger type sizes, such as those used for ads, headlines, and other features, were not supported by the linotype and its banks of molds. These fell to nimble fingers to choose, arrange, and align individual characters from the appropriate trays with a speed that might well have been envied by even expert typists.

Still, by the time I graduated from high school (and, at the same time, completed a 12-year apprenticeship in the mysteries of the newspaper business), it was clear that, soon—at least in technologically historical terms—both the gentle monster and the type trays would be little more than museum exhibits. It was not too many years later that both did, indeed, disappear. They were replaced, first, by electronic/optical/photographic processes and then, a scant decade after that, under my own supervision (as a visiting computer consultant), by purely electronic processes.

Today, of course, these are only the memories of a one-time printer's devil. But, even if the old order has passed, the type tray and linotype laid the foundations for the modern world. They are reflected not only in modern fonts and typesyles, but also in the terminology that defines font characteristics and in the methods that manipulate their appearance.

A Brief History of Typefaces

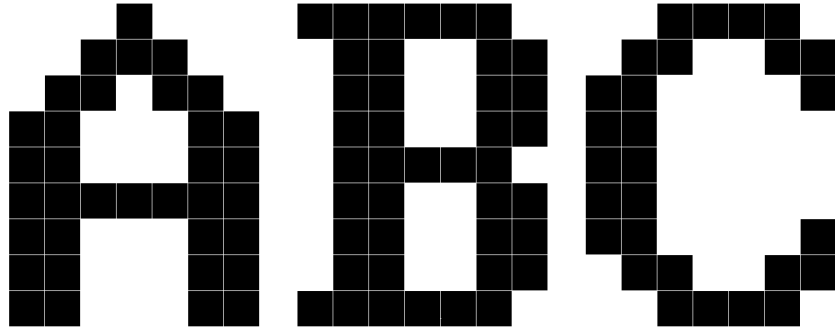
When computers were young, typefaces were an embellishment limited to high-end, hard-copy devices, such as daisywheel printers; even then, they were changed only by physically changing the print wheel. For computer monitors, type styles were quite simply firmware built into the system. In general, they consisted of 8×8 or 8×9 bitmapped (also called *raster*) characters for CGA video systems. These ranged up to 8×18 bitmapped characters for VGA systems.

Bitmapped Fonts

Figure S14.1 shows three bitmapped characters in an 8×12 format. Bitmapped fonts have some obvious advantages. Since each character's pixel image is already defined, the character can be transferred to the screen by simply copying the bit pattern directly to the video. This process is speedy and places minimal demands on system resources.

FIGURE S14.1:

Bitmapped fonts



However, there are disadvantages to using bitmapped fonts. They can be resized only as simple multiples and cannot be created in any in-between sizes. When enlarged, the resulting characters tend to be jagged in appearance.

Also, while some systems did offer more than one font, the selection was generally limited to two or three sizes, such as font provisions for a 43- or 50-line display as alternatives to the standard 25 lines, but without offering any variations in style, pitch, or weight. Of course, on early computers, there was little or no demand for larger typefaces or even for varying typefaces. It remained for the advent (and popularity) of graphics display systems to demonstrate the advantages of sizable fonts.

Stroked Fonts

One early approach to creating fonts for a graphics environment involved creating libraries of bitmapped fonts in incremental sizes. As a solution, however, this was never popular for several reasons: because of the sheer mass of data required for the fonts, because of relatively slow response times, and because of the demands on the system memory.

Instead, a different way to define characters was devised (or, more accurately, borrowed from existing typesetting technologies already in use by printers in the newspaper and publishing industries). These are known as *stroked* or *vectored* fonts. In this system, the structure of each character is described by a series of vectors, not by an array of pixels.

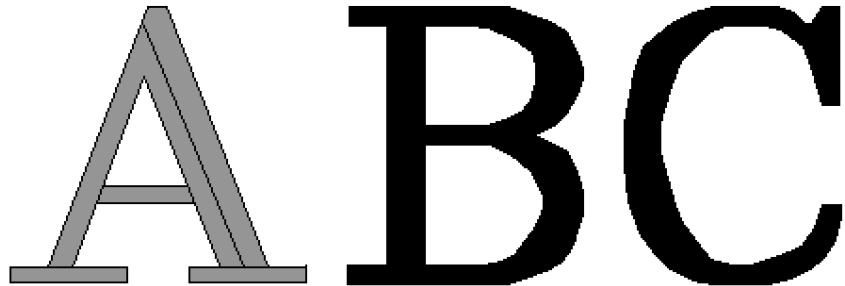
There are some disadvantages to the stroked font approach. For small font sizes, the vectored data is, in general, larger than an equivalent bitmapped font and requires more processing to produce each display character. But the disadvantages are minor, placing only minimal demands on modern CPUs and contemporary video systems.

The advantages are tremendous. Stroked fonts can not only be resized, but they can also be reproportioned, weighted, slanted, rotated, inverted, or otherwise manipulated with minimal effort and maximal effect. And, most important, a single set of font data provides a variety of sizes and styles within a single typeface. Once a font is defined as stroked data, the resulting typeface is available in any size desired—as italics, boldface, or with sufficiently sophisticated processing, as outline, condensed, or extra-bold forms.

Figure S14.2 shows three characters created using a vectored font, sized for 48 points. The *A* shows the vectors defining the character as black lines. The *B* and *C* characters show the outlines after processing.

FIGURE S14.2:

A stroked, or vectored, font



Under Windows 95, 98, NT, and 2000, the older, bitmapped fonts have been largely (but not entirely) discarded in favor of stroked fonts.

Windows Default Fonts

Windows supplies the 15 standard fonts shown in Figure S14.3. Each of these fonts appears in its default height, width, and weight.

TABLE S14.4 (CONTINUED): Windows Standard Fonts

Font	Height	Avg Width	Comments
Script	18	8	Font that resembles handwriting (appears small for point size)
Small Fonts	11	5	Small sans-serif font; good for readable fine print
Symbol	16	8	Greek, math, and other symbols
System	16	7	Proportional-width system font (sans-serif)
Terminal	16	13	A rather broad sans-serif font
Times New Roman	17	6	Popular proportional-width, general-purpose font
Wingdings	16	13	Useful symbols; also called Dingbats

The Courier, Fixedsys, MS Sans Serif, MS Serif, Small Fonts, System, and Terminal fonts are essentially bitmapped fonts.

Of the remaining eight fonts, the Modern, Roman, and Script fonts consist only of strokes. When they are drawn as enlarged characters (for example, at a height of 400 in text mode), they quite clearly show the strokes comprising each character in a fashion similar to the stroked *A* in Figure S14.1.

The other five fonts—Arial, Courier New, Symbol, Times New Roman, and Wingdings—are stroked outline (or True-Type) fonts. Stroked outline fonts are created as outline strokes with the interiors filled. When these are drawn in larger sizes, they remain fully solid, even though their outlines may begin to show a slight grain or irregularity.

NOTE

The fonts distributed with the release version of Windows 2000 may be different from those listed above, and individual fonts are being changed from stroked to full True-Type fonts. Also, since you may have fonts installed on your system from several different sources, the comments on specific fonts should be taken only as a general guideline.

Font Selection Using Logical Fonts

While you might think of font selection as simply being a matter of requesting a typeface and specifying a character size, for computers, this is a bit too simple. This is not because computers require complexity, but because a font—even a sizable font—still must match the display characteristics of the device, at least to a minimal degree.

Thus, instead of simply naming a typeface and size, an application makes a request identifying the font name, size, and other characteristics desired. For this purpose, the LOGFONT structure is defined in WinGDI.H as:

```
typedef struct tagLOGFONTA
{
    LONG lfHeight;
    LONG lfWidth;
    LONG lfEscapement;
    LONG lfOrientation;
    LONG lfWeight;
    BYTE lfItalic;
    BYTE lfUnderline;
    BYTE lfStrikeOut;
    BYTE lfCharSet;
    BYTE lfOutPrecision;
    BYTE lfClipPrecision;
    BYTE lfQuality;
    BYTE lfPitchAndFamily;
    char lfFaceName[LF_FACESIZE];
} LOGFONTA;
```

Two structure definitions are provided: LOGFONTA, for use with an ANSI environment, and LOGFONTW, for use with a wide, or Unicode, character set. However, since the choice of environment is controlled by a compiler directive, the only source code reference required is LOGFONT. Depending on the compiler directive, this will be mapped to either the ANSI or Unicode structure, as appropriate.

The following sections describe the fields in the LOGFONT structure.

Height and Width Fields

The lfHeight field specifies the desired height of the font in logical units. If the value is positive or negative, the absolute value is transformed to device units and matched against the cell heights of the available fonts.

A 0 (zero) height simply instructs the GDI to select a reasonable (default) height. This is normally the smallest size that will accommodate the strokes comprising the font characters.

In all cases, the font mapper looks for the largest font—the most detailed font (character) description—that does not exceed the requested size. If none match this requirement, the smallest available font is used.

The `lfWidth` field specifies the average width (in logical units) of the characters in the font. If `lfWidth` is 0, the device aspect ratio is matched against the digitization aspect ratio; that is, the width in units which the font will require for display. Selection is based on the closest match, or the smallest absolute difference between the two ratios. In general, a 0 value allows the character width to be matched against the character height.

In actual practice, bitmapped fonts, such as the Courier, Fixedsys, MS Sans Serif, MS Serif, Small Fonts, System, and Terminal fonts (illustrated in Figure S14.3 and listed in Table S14.4) are used, as long as the bitmap size matches the display context relatively well. If any of these are enlarged, however, Windows substitutes a default stroked font, usually Arial, which can be more readily sized. (You can see this in action in the *Fonts* demo, discussed later in the chapter.)

Weight Field

The `lfWeight` field specifies the desired weight of the font. This field accepts values in the range 0 to 1000. If `lfWeight` is 0, a default weight is used (normal). As you can see in Table S14.5, currently only two weights are actually employed: 400 for normal or 700 for bold. However, future versions (probably with higher-resolution displays) are expected to use a wider range of weights.

TABLE S14.5: Font Weights

Weight Constant	Value	Comments	Alternatives
FW_DONTCARE	0		
FW_THIN	100	Not supported	
FW_EXTRALIGHT	200	Not supported	FW_ULTRALIGHT
FW_LIGHT	300	Not supported	
FW_NORMAL	400	Default weight	FW_REGULAR

Continued on next page

TABLE S14.5 CONTINUED: Font Weights

Weight Constant	Value	Comments	Alternatives
FW_MEDIUM	500	Not supported	
FW_SEMIBOLD	600	Not supported	FW_DEMIBOLD
FW_BOLD	700	Boldface	
FW_EXTRABOLD	800	Not supported	FW_EXTRABOLD
FW_HEAVY	900	Not supported	FW_BLACK

Italic, Underline, and Strikeout Fields

The `lfItalic` field is simply a Boolean flag. If TRUE, the font is created as italics (if possible). The `lfUnderline` and `lfStrikeOut` fields operate in the same fashion.

Character Set Field

The `lfCharSet` field specifies the character set desired. Identifiers are predefined in `WinGDI.H`, as shown in Table S14.6.

TABLE S14.6: Character Set Constants

Character set	Value	Comments
ANSI_CHARSET	0	Default; ANSI characters
UNICODE_CHARSET	1	Unicode (32-bit) characters
SYMBOL_CHARSET	2	Symbols
SHIFTJIS_CHARSET	128	Japanese Kanji characters
HANGEUL_CHARSET	129	Non-Roman/Arabic characters
CHINESEBIG5_CHARSET	136	Chinese characters
OEM_CHARSET	255	Device-dependent characters

The following character sets are defined only for WinVer 0x0400 (Windows 95/98/NT) & later.

JOHAB_CHARSET	130	
HEBREW_CHARSET	177	Hebrew (Judaic) characters

Continued on next page

TABLE S14.6 (CONTINUED): Character Set Constants

Character set	Value	Comments
The following character sets are defined only for WinVer 0x0400 (Windows 95/98/NT) & later.		
ARABIC_CHARSET	178	Arabic characters
GREEK_CHARSET	161	Greek characters
TURKISH_CHARSET	162	Turkish characters
VIETNAMESE_CHARSET	163	Vietnamese characters
THAI_CHARSET	222	Thai (Thailand) characters
EASTEUROPE_CHARSET	238	Eastern European characters
RUSSIAN_CHARSET	204	Russian characters
MAC_CHARSET	77	Macintosh characters
BALTIC_CHARSET	186	Baltic (region) characters
FS_LATIN1	0x00000001L	Latin characters
FS_LATIN2	0x00000002L	Latin characters
FS_CYRILLIC	0x00000004L	Russian characters
FS_GREEK	0x00000008L	Greek characters
FS_TURKISH	0x00000010L	Turkish characters
FS_HEBREW	0x00000020L	Hebrew (Judaic) characters
FS_ARABIC	0x00000040L	Arabic characters
FS_BALTIC	0x00000080L	Baltic (region) characters
FS_VIETNAMESE	0x00000100L	Vietnamese characters
FS_THAI	0x00010000L	Thai (Thailand) characters
FS_JISJAPAN	0x00020000L	Japanese characters
FS_CHINESESIMP	0x00040000L	Chinese characters
FS_WANSUNG	0x00080000L	Chinese characters
FS_CHINESETRAD	0x00100000L	Chinese characters
FS_JOHAB	0x00200000L	
FS_SYMBOL	0x80000000L	Symbol font

WARNING Although fonts supporting character sets other than those defined may be present in a system, do not attempt to translate or interpret strings to be rendered with such fonts.

Escapement and Orientation Fields

Both the `lfEscapement` and `lfOrientation` fields are expressed in 1/10° increments. The `lfEscapement` value sets the string orientation with an angle of 0° for horizontal alignment, increasing in a counter-clockwise direction.

The `lfOrientation` value determines the angle of the character’s baseline relative to horizontal. Thus, for a value of 0, a *T* or *L* remains vertical; for a value of 900 (90°), *T* will be drawn horizontally and the *L* will be lying on its back.

Table S14.7 summarizes both the text and character orientation at 90° intervals for `lfEscapement` and `lfOrientation`.

TABLE S 14.7: Text and Character Orientation

Value	Degrees	IfEscapement (String Orientation)	IfOrientation (Character Orientation)
0	0°	Left to right (default)	Normal (vertical, default)
900	90°	Vertical, rising	Rotated 90° counter-clockwise
1800	180°	Right to left	Inverted
2700	270°	Vertical, falling	Rotated 90° clockwise

Out-Precision, Clip-Precision, and Quality Fields

The `lfOutPrecision`, `lfClipPrecision`, and `lfQuality` fields are used to request specific matches between the fonts selected and the device-output capabilities.

`lfOutPrecision` defines how closely the actual output must match the requested font’s characteristics, such as height, width, orientation, and pitch. Output precision values are defined as shown in Table S14.8.

TABLE S14.8: Output Precision

Constant	Value	Comments
OUT_DEFAULT_PRECIS	0	
OUT_STRING_PRECIS	1	Maintain string precision
OUT_CHARACTER_PRECIS	2	Maintain character precision
OUT_STROKE_PRECIS	3	Maintain stroke precision
OUT_TT_PRECIS	4	New, not documented; support unknown
OUT_DEVICE_PRECIS	5	New, not documented; support unknown
OUT_RASTER_PRECIS	6	New, not documented; support unknown
OUT_TT_ONLY_PRECIS	7	New, not documented; support unknown
OUT_OUTLINE_PRECIS	8	Maintain outline precision

NOTE

Several of the flag values for the `lfOutPrecision`, `lfClipPrecision`, and `lfQuality` fields are new and may or may not be fully supported by present versions of Windows 98 and/or by present video and output device drivers. Before you rely on a specific precision flag, you should experiment with it. Unimplemented precision flags may be supported later or may be supported by specific device drivers.

The `lfClipPrecision` field specifies how characters that are partially outside the clipping region are clipped. Eight values are defined in `WinGDI.H`, as shown in Table S14.9.

TABLE S14.9: Clipping Precision

Constant	Value	Comments
CLIP_DEFAULT_PRECIS	00h	
CLIP_CHARACTER_PRECIS	01h	Clip entire character
CLIP_STROKE_PRECIS	02h	Clip only strokes
CLIP_MASK	0Fh	New, not documented; support unknown

Continued on next page

TABLE S14.9 CONTINUED: Clipping Precision

Constant	Value	Comments
CLIP_LH_ANGLES	10h	New, not documented; support unknown
CLIP_TT_ALWAYS	20h	New, not documented; support unknown
CLIP_EMBEDDED	80h	New, not documented; support unknown

The `lfQuality` field specifies the desired output quality, which is how well the output (physical font) is matched to the requested logical font attributes. Three values are defined in `WinGDI.H`, as shown in Table S14.10.

TABLE S14.10: Output Quality

Constant	Value	Comments
DEFAULT_QUALITY	00h	Appearance not important
DRAFT_QUALITY	01h	Appearance of minimal importance; font scaling fully enabled for all GDI fonts; bold, italic, underline, and strikeout synthesized as necessary
PROOF_QUALITY	02h	Character quality more important than matching logical font attributes; GDI font scaling disabled; only closest font sizes chosen; bold, italic, underline, and strikeout synthesized as necessary

Pitch and Family Field

The `lfPitchAndFamily` field specifies both the pitch (spacing) and the font family. The two low-order bits specify the font spacing, using one of the three values defined in `WinGDI.H`, as shown in Table S14.11.

TABLE S14.11: Font Pitch

Pitch Constant	Value	Comments
DEFAULT_PITCH	00h	Don't care or don't know
FIXED_PITCH	01h	Fixed spacing (characters per inch)
VARIABLE_PITCH	02h	Variable spacing (proportional)

The high-order nibble of the `lfPitchAndFamily` byte specifies a family of fonts and can be any of the six values defined in `WinGDI.H`, as shown in Table S14.12.

TABLE S14.12: Font Family

Family Constant	Value	Comments
<code>FF_DONTCARE</code>	00h	Don't care or don't know
<code>FF_ROMAN</code>	10h	Serif, variable character width, such as Times Roman and Century Schoolbook
<code>FF_SWISS</code>	20h	Sans-serif, variable character width, such as Helvetica and Swiss
<code>FF_MODERN</code>	30h	Constant character width, serif or sans-serif, such as Pica, Elite, and Courier
<code>FF_SCRIPT</code>	40h	Cursive, for example
<code>FF_DECORATIVE</code>	50h	Old English, for example

Face Name Field

The `lfFaceName` field contains the address of a null-terminated string specifying the typeface name of the desired font. The string must not exceed 32 characters. If no font name is specified (the argument is `NULL`), the GDI uses a default typeface such as Arial.

NOTE

For typesetting purposes, WinVer 0x0400 (Windows 95/98/NT) and later also support a number of additional font specifications, including an extensive series of Panose definitions, which are not discussed here. The Panose font-classification values are contained in a `PANOSE` structure and describe the characteristics of a True Type font. For further information, refer to the online documentation.

The *Fonts* Demo: Demonstrating Logical Fonts



The *Fonts* demo provides a platform to demonstrate the three principal features of using logical fonts:

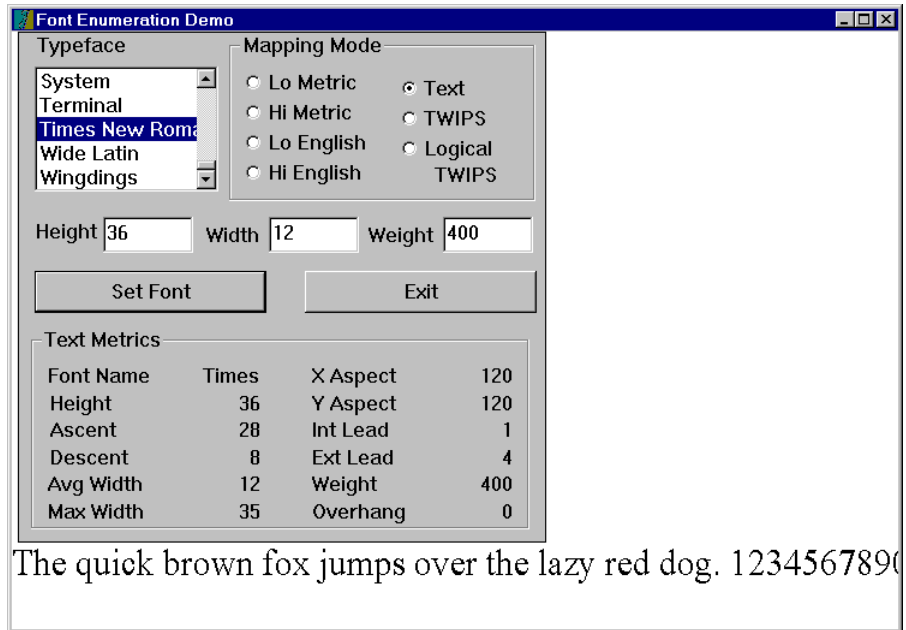
- Using the `EnumFonts` function to list available typefaces
- Setting font characteristics (height, width, and so on)

- Showing fonts under different mapping modes

Figure S14.4 shows the *Fonts* demo window.

FIGURE S14.4:

The *Fonts* demo



A standard font-selection dialog box is supplied through the MFC `CFontDialog` class as one of the Windows common dialog boxes. The advantage of using the standard Fonts dialog box is that you do not need to provide your own font-selection mechanisms. (For an example of the common font dialog box, refer to the *FontView* demo included with the Visual C++ compiler.)

NOTE

The *Fonts* demo is included on the CD that accompanies this book, in the Supplement 14 folder. This listing should provide you with the basic structure for designing your own font-manipulation facilities.

Font Selection

The first feature demonstrated in the *Fonts* demo is the use of the `EnumFonts` function to query the GDI and list all available typefaces, in this case, by loading the font names in a list box.

As a first step, we define the record type FONTLIST:

```
typedef struct tagFONTLIST
{
    GLOBALHANDLE  hGMem;
    int           nCount;
} FONTLIST;
```

A variable of type FONTLIST is used to record a list of fonts and the number of fonts located.

In most of the other demos discussed in this book, the `WndProc` procedure provides the heart of the application. In addition to message handling, this procedure is responsible for a greater or lesser portion of the application's task handling. In contrast, in the *Fonts* demo, the majority of the work is shifted away from the `WndProc` procedure to a modeless dialog box and to the `DlgProc` procedure-handling messages addressed to the dialog box. The few provisions necessary in `WndProc` for creating `DlgProc` should, at this point, already be familiar to you from previous examples.

The exported `DlgProc` procedure begins by declaring two static variables. One of these is `lpEnumProc`, a pointer to the `FontEnumFunc` procedure, which will actually make the call to the API `EnumProc` function. The other is a `FontList` variable to point to the returned data (the typeface names). A third variable is declared as a long pointer to a string, `lpFontName`, providing a second handle to the font list.

```
BOOL APIENTRY DlgProc( HWND hDlg, UINT msg, UINT wParam, LONG lParam )
{
    static FARPROC lpEnumProc;
    static FONTLIST FontList;
    LPSTR          lpFontName;
    HDC             hdc;
    int             i, nSel;
    char            szFont[LF_FACESIZE];
```

In response to the `WM_INITDIALOG` message, another call is made to the `MakeProcInstance` macro to return a handle (`lpEnumProc`) to the `FontEnumFunc` procedure. This handle is used to tell the `EnumFonts` function where to find the `FontEnumFunc` callback function.

```
switch( msg )
{
    case WM_INITDIALOG:
        ...
        lpEnumProc = MakeProcInstance( FontEnumFunc, hInst );
```

```
FontList.hGMem = GlobalAlloc( GMEM_MOVEABLE |
                             GMEM_ZEROINIT, 1L );
FontList.nCount = 0;
```

The `FontList` variable also requires initialization by allocating and zeroing one (1) byte, returning the memory pointer to the `FontList.hGMem` field. Additional memory will be allocated as required, but since the number of fonts is not presently known, it would be pointless to attempt to allocate memory for an unknown number of strings at this time. The `FontList.nCount` field is also initialized to 0 but will be used presently to track the number of typefaces found.

The next requirement is a device-context handle (`hdc`), which is obtained using `GetParent(hDlg)` to return a device context for the main application window rather than the dialog window. And, finally, the `EnumFonts` API function is called using the retrieved device-context handle, a pointer to the `FontEnumProc` function, and a pointer to the `FontList` variable.

```
hdc = GetDC( GetParent( hDlg ) );
EnumFonts( hdc, NULL, 1pEnumProc, (LPVOID) &FontList );
```

The second parameter, passed as `NULL` in this example, could have been used as a long pointer to a string (`LPSTR`) to specify a particular typeface; in effect, to query if a specific typeface was available. By passing this specification as `NULL`, all available typefaces will be reported.

The `FontEnumProc` procedure, while not an exported procedure in the usual sense of receiving messages directly from Windows, is used as a callback function by the `EnumFonts` API function.

The `FontEnumProc` procedure is called from `EnumFonts` with four parameters: a pointer to a `LOGFONT` structure reporting the logical attributes of a font, a pointer to a `TEXTMETRIC` structure reporting the physical font attributes, a short integer indicating the font type (bit flags), and the same far pointer to `FontList` that was originally passed to `EnumFonts`.

```
int APIENTRY FontEnumFunc( LPLOGFONT    lf,
                          LPTEXTMETRIC  tm,
                          short          nFontType,
                          FONTLIST FAR * FontList )
{
    LPSTR 1pFontFace;
```

The first task accomplished in the `FontEnumFunc` procedure is a `GlobalReAlloc` to allocate enough additional memory for one more typeface name. If this is successful, `GlobalLock` is called to ensure that the memory allocated is not moved until the present task is finished.

```

    if( ! GlobalReAlloc( FontList->hGMem,
                        (DWORD) LF_FACESIZE * ( FontList->nCount + 1 ),
                        GMEM_MOVEABLE ) )
        return( FALSE );
    lpFontFace = GlobalLock( FontList->hGMem );
    lstrcpy( lpFontFace + ( ( FontList->nCount ) * LF_FACESIZE ),
            (LPSTR) lf->lfFaceName );
    GlobalUnlock( FontList->hGMem );
    FontList->nCount++;
    return( TRUE );
}

```

Once memory is allocated and locked, the font name (`lf->lfFaceName`) reported by `EnumFonts` is copied to the offset of the newly allocated memory, the memory is unlocked, and `FontList->nCount` is incremented.

NOTE

As you may have already realized, the bulk of the information passed by `EnumFonts` has been discarded; only the font name has been retained. But, in this case, the font name is all that we really need. Actually, there wasn't any choice about what information was passed to the `FontsEnumFunc` procedure from the `EnumFonts` API function. The only choice was to select which portion was actually wanted, discarding the excess. (We'll talk about another alternative in a bit.)

The next step is to copy the font list into the list box in the dialog box. And, again, this process begins by globally locking the memory where the data is stored.

```

lpFontName = GlobalLock( FontList.hGMem );
SendDlgItemMessage( hDlg, IDD_FONTLIST, WM_SETREDRAW,
                    (WPARAM) 1, (LPARAM) 0 );
SendDlgItemMessage( hDlg, IDD_FONTLIST,
                    LB_RESETCONTENT, NULL, NULL );

```

Before copying the data, however, as a precaution, two initial messages are sent to the list box to instruct it to be redrawn. These ensure that the new data will be visible. Also, a reset message is sent to clear any contents that the list box might happen to contain.

Once this housekeeping is out of the way, a simple loop, using `FontList.nCount` as the limit, copies the string data. The names are copied one item at a time, using the same offset addresses as before, into the list box.

```

for( i=0; i<FontList.nCount; i++ )
    SendDlgItemMessage( hDlg, IDD_FONTLIST, LB_ADDSTRING, 0,
        (LPARAM)(LPSTR)( lpFontName + ( i * LF_FACESIZE ) ) );
GlobalUnlock( FontList.hGMem );
GlobalFree( FontList.hGMem );
FontList.hGMem = NULL;
FreeProcInstance( lpEnumProc );
ReleaseDC( GetParent( hDlg ), hdc );

```

After the data has been copied over to the list box, the allocated memory (hGMem) and the address variable for the FontEnumProc procedure are no longer required. Therefore, the allocated memory is unlocked and freed, the lpEnumProc function handle is freed, and to finish the housekeeping, the device context is released.

Finally, after calling the ShowMetrics function to update the dialog box display, the WM_INITDIALOG response, rather than returning on a break; statement, is allowed to fall through to the WM_SETFOCUS message response, setting the active focus to the dialog box.

```

    ShowMetrics( hDlg ); // fall through to SetFocus

case WM_SETFOCUS:
    SetFocus( GetDlgItem( hDlg, IDD_HEIGHT ) );
    break;

```

At this point, the list box in the dialog box is primed with a list of available fonts, ready for the user to select the typeface desired. And, when this is done, the dialog box display will be updated to reflect the selection. The main application window will display the sample text string using the chosen typeface.

TIP

If the process of retrieving the font list seems rather roundabout, a portion could be simplified. For instance, what about rewriting the FontEnumFunc callback function to load the list box directly? This would simplify matters, wouldn't it? This is an experiment that you may want to try.

Font Characteristic Variations

In addition to allowing the user to select typefaces from the list box, the *Fonts* demo also provides three edit boxes for entry of height, width, and weight specifications. Values entered in these three edit boxes are used (when the Set Font button is clicked) in the tm (text metric) request that selects the new font (or new size).

The resulting text metrics information items reported at the bottom of the dialog box (look back at Figure S14.4) represent the best match found by the GDI and reflect the actual font displayed in the application's main window (immediately below the dialog box).

TIP

In the Font demo's current form, only four of the text metrics fields can be edited directly or indirectly; these are the height, width, weight, and typeface. The remaining fields are assigned default values. As another experiment, you could add control features for any or all of the other text metrics.

Font Sizing and Mapping Modes

The *Fonts* demo also includes a provision for selecting the different mapping modes. This part of the program demonstrates how different fonts appear when resized, how fonts can be changed proportionally by varying the width and height, and how the GDI selects fonts appropriate to the mapping mode and sizes requested. (For more information about mapping modes, see Supplement 13.)

Notice particularly that, when one of the bitmapped fonts is requested in a too-large size, the GDI replaces it with a stroked or outline font, which is more suitable for the purpose.

TIP

You may wish to change the defaults assigned to the `lfPitchAndFamily` specification and observe how this affects the GDI's choices. Last, for the adventurous or the dedicated, the `lfEscapement` and `lfOrientation` fields offer ample opportunities for departure from the straight and horizontal. Please feel free to explore the possibilities.

The real test of typeface flexibility lies in the applications using these facilities. Any shortcomings are more likely to lie in the application design than in the provided resources.

As you have seen, the basic tools permit virtually any degree of text elaboration desired. For those requiring additional typefaces, a variety of fonts are available from third-party sources. You can also find toolkits for designing custom fonts.

Now that we've covered the advanced text operations, we can return to graphics operations. The next chapter discusses images and image file formats.

Graphics Utilities and File Operations

- A screen-capture program for bitmap images
- Bitmap compression techniques
- Commands for handling graphics files
- A PCX file viewer
- Techniques for converting 24-bit color images

A variety of formats have been developed for saving, storing, and displaying graphics images. A number of popular formats predated Windows, including ZSoft's Paintbrush PCX, CompuServe's GIF and, for more demanding circumstances, TrueVision Inc.'s TARGA or TGA formats. Today, Windows has its own native BMP format.

All of these formats have one factor in common: Each is designed for a specific image type or system. TrueVision's TGA images, for example, are designed for video-camera images, generally incorporating 16, 24, or 32 bits of color per pixel. In contrast, ZSoft's PCX, CompuServe's GIF, and Windows BMP formats are palette-based, encoding images by first including palette-color information in the image file and then referencing the individual pixels as palette colors.

In this chapter, we'll take a look at how to handle graphics files. We'll begin by discussing a demo program that captures Windows bitmap images either to the clipboard or to a file. Then we'll cover commands for handling graphics files, treatments for file formats other than the native BMP files, and techniques for converting 24-bit color images.

A Screen-Capture Utility

Under Windows, where all displays are graphical in nature, a graphics screen-capture utility can be a basic tool for transferring graphics information among applications or for simply saving images for later use. Capturing a screen under Windows is greatly expedited and simplified by procedures inherent within Windows.

Principal among these methods is the Windows clipboard, a facility which provides both storage and information transfer among Windows applications. (Although the clipboard handles several types of information, each with its own format, only the graphics image or bitmap format is relevant to this discussion.)

Capturing a bitmap to the clipboard is a task well-supported by Windows API functions, making this task almost automatic. In contrast, a similar capture to a file format requires several steps. Procedures for both capturing to the clipboard and to a file are included in the *Capture* demo.

The *Capture* Demo: Capturing and Displaying Screen Images



The *Capture* demo is a screen-capture utility that stores and retrieves only information that is in bitmap format. Other types of information loaded to the clipboard—such as data, metafiles, or other formats—are ignored.

As an example of the use of the clipboard to store a bitmap, take a look at Figure S15.1. This figure was created using the *Capture* demo.

Figure S15.1 is neither a trick nor a composite. It's an actual screen display, which was created as follows:

1. Begin by loading two instances of the *Capture* demo. Initially, one image is reduced to an icon, and the second is used to capture the screen to the system clipboard. During the capture process, the second instance automatically reduces itself to an icon at the bottom of the screen; then it resumes normal size after the capture is completed.
2. After the first capture, both copies of *Capture* are restored to windows. Both show the clipboard image using the Fit to Window option.

FIGURE S15.1:

Recursively captured views



3. One copy of *Capture* is used to repeat the process of capturing the screen, including the image of the other copy of *Capture* and the previous clipboard image. Each new image is automatically written to the clipboard and then displayed by both copies of *Capture*.
4. Repeating this process yields the recursive image shown in Figure S15.1, where it becomes a tunnel effect.
5. Finally, call the last capture operation to capture the recursive image, but this time, rather than capturing the image to the clipboard for display, write the image directly to a .BMP file. The instance making the capture reduces itself to an icon during capture. Therefore, that instance loses the active focus, which reverts to the open window showing the previous clipboard image.

Because the final operation saves the image to a file rather than to the clipboard, the previously displayed image is not replaced. If a different utility were used to capture an image to the clipboard, both of the *Capture* windows would display that image.

Both capture options in the *Capture* demo—To Clipboard and To File—include a five-second time delay (audible beeps are sounded at one-second intervals during the wait time), during which the user may use the mouse or keyboard to shift the active focus, to select a different application window, to pull down menus, or to activate other features.

NOTE

The Capture demo is included on the CD that accompanies this book, in the Supplement 15 folder.

Screen-Capture Operations

Screen capture is based on rectangular coordinates that define the area to be copied. For the example, the *Capture* demo simply checks the size of the Desktop (the main screen), using the `HWND_DESKTOP` window handle, to capture the entire display. Other screen-capture applications could include provisions to select only the active application's display, to select only a specific window, or to use the mouse to select some other rectangular area.

In operation, because the Capture menu offers the To Clipboard and To File choices, two responses are provided for the WM_COMMAND/IDM_CLIP and WM_COMMAND/IDM_FILE messages:

```
case WM_COMMAND:
    switch( LOWORD( wParam ) )
    {
        ...
        case IDM_CLIP:
            Action = TOCLIPBD;
            CloseWindow( hwnd );
            Clock = SetTimer( hwnd, 1, 1000, NULL );
            iSec = 0;
            break;

        case IDM_FILE:
            if( DialogBox( hInst, "GETNAME", hwnd,
                          FileNameDlgProc ) )
            {
                Action = TOFILE;
                CloseWindow( hwnd );
                Clock = SetTimer( hwnd, 1, 1000, NULL );
                iSec = 0;
            }
            break;
```

The two responses are essentially the same. The difference is that before an image can be saved to a file, the `FileNameDlgProc` is invoked, requesting a filename and, optionally, a drive and path specification.

In both responses, the `CloseWindow` API is called to minimize the *Capture* application before initializing a timer for one-second intervals and setting the seconds counter (`iSec`) to zero.

Subsequently, as WM_TIMER messages are received, `iSec` is incremented. Until `iSec` reaches five seconds (or whatever interval is desired), the `MessageBeep` function is called to provide an audible timer signal.

```
case WM_TIMER:
    if( ++iSec == 5 )
        PostMessage( hwnd, WM_COMMAND, IDM_CAPTURE, 0 );
    else
        MessageBeep( MB_ICONEXCLAMATION );
    break;
```

NOTE

The sound (waveform) associated with each `MB_ICONxxxxx` constant may be changed using the Sound Control Panel. These assignments, however, are under the control of the user, not the application.

The `MB_ICONEXCLAMATION` argument used here in calling the `MessageBeep` API function is probably more familiar as an argument in a `MessageBox` API call requesting inclusion of the exclamation icon. However, the `MB_ICONEXCLAMATION` constant, as well as the `MB_ICONASTERISK`, `MB_ICONHAND`, `MB_ICONQUESTION`, and `MB_OK` constants, can also be used as parameters to request a system sound that is defined as a .WAV waveform file and reproduced by a sound card, such as Sound Blaster. If no sound card is installed, the system speaker will provide the traditional default beep using the system's internal speaker.

Next, when the `IDM_CAPTURE` message is received, a final beep is issued and the timer process is halted (killed) before either the `SaveBitmap` function is called to create a bitmap file or the `CaptureBitmap` function is called to copy the image to the clipboard.

```
case IDM_CAPTURE:
    MessageBeep( MB_OK );
    KillTimer( hwnd, Clock );
    SetCursor( LoadCursor( NULL, IDC_WAIT ) );
    switch( Action )
    {
        case TOFILE:
            SaveBitmap();          break;
        case TOCLIPBD:
            CaptureBitmap( hwnd ); break;
    }
    SetCursor( LoadCursor( NULL, IDC_ARROW ) );
    OpenIcon( hwnd );
    break;
```

Once the appropriate process is completed, the `OpenIcon` API function is called to restore the application to its original window size and state. At this time, the application also regains the active focus.

The two capture processes—`CaptureBitmap` and `SaveBitmap`—use parallel operations but are not identical.

Clipboard-Capture Operations

The CaptureBitmap process begins by calling the GetDC API function but, instead of using the application's window handle (hwnd), we use the Desktop handle (HWND_DESKTOP).

```
int CaptureBitmap( HWND hwnd )
{
    HDC          hdc, hdcMem;
    HBITMAP      hBitmap;
    static int    i, j, CRes, LnWidth, LnPad = 0,
                 xSize, ySize;

    SetCursor( LoadCursor( NULL, IDC_WAIT ) );
    hdc = GetDC( HWND_DESKTOP );
    xSize = GetDeviceCaps( hdc, HORZRES );
    ySize = GetDeviceCaps( hdc, VERTRES );
```

Once we have retrieved a handle to the Desktop device context, we can use the GetDeviceCaps function to query the current display resolution. To capture only a specific application's window, the parallel process would be to simply use the application's handle; for example, use the GetFocus API function to retrieve a handle for the active application's window.

The next step is to create a memory context that is compatible with the selected device context, and then to create a compatible bitmap.

```
hdcMem = CreateCompatibleDC( hdc );
hBitmap = CreateCompatibleBitmap( hdc, xSize, ySize );
```

Notice that the compatible bitmap created here is not a display bitmap. Rather, hdcMem is a memory device-context handle that contains a copy of the display image. This way, we can manipulate the image data without affecting the actual display.

The next step includes a provisional test before proceeding to ensure that a valid bitmap handle was returned. If the call to CreateCompatibleBitmap is successful, hBitmap will be non-NULL, and the capture process can proceed by calling SelectObject to select the bitmap, hBitmap, into the logical context, hdcMem. However, it is the subsequent StretchBlt instruction that actually transfers the image from the screen to the memory device-context handle.

```
if( hBitmap )
{
    SelectObject( hdcMem, hBitmap );
    StretchBlt( hdcMem, 0, 0, xSize, ySize,
                hdc,    0, 0, xSize, ySize, SRCCOPY );
```

Although both `BitBlt` and `StretchBlt` provide a means of copying information between display contexts (`hdc` and `hdcMem` in this example), `StretchBlt` optionally provides the additional capability of stretching (or shrinking) the image to fit the available display space. The `BitBlt` function simply executes an exact copy.

NOTE

Notice that even though `StretchBlt` is being used for the transfer, the source and destination rectangles are the same size. Thus, no distortion is imposed on the image being copied. The `StretchBlt` operation is necessary to copy the image pixels from the active device context to the memory context, where they become the bitmap referenced by the `hBitmap` handle.

The memory device context, however, is not the actual destination for this bitmap. Rather, `hdcMem` is used as an environment where the copy of the original image can be stretched or shrunk to fit the application's display context.

Copying to the Clipboard

We want to give the bitmap a more permanent storage location and make it accessible to other applications. Calling `OpenClipboard` opens the clipboard for examination. The next instruction, `EmptyClipboard`, gives the current application temporary ownership of the clipboard, dumping the current contents (if any) of the clipboard.

```
OpenClipboard( hwnd );  
EmptyClipboard();  
SetClipboardData( CF_BITMAP, hBitmap );  
CloseClipboard();
```

Next, we call `SetClipboard` to transfer the new material—the bitmap image—to the clipboard, specifying the data type with the `CF_BITMAP` argument. Then we call `CloseClipboard`, releasing ownership of the clipboard.

NOTE

The preceding sequence is a fairly standard example of clipboard use, beginning with `OpenClipboard` and ending with `CloseClipboard`. Between the `Open...` and `Close...` commands, a variety of different actions can be executed. But, remember that control (ownership) of the clipboard is always temporary and should be relinquished as quickly as possible. See Chapter 20 in the book for more information about clipboard operations.

Finally, as with any process, a degree of cleanup is required. It begins, still within the conditional process, by invalidating the application's window to ensure that the application will be repainted after it is restored:

```
        InvalidateRect( hwnd, NULL, TRUE );
    }
    DeleteDC( hdcMem );
    ReleaseDC( HWND_DESKTOP, hdc );
    return 0;
}
```

The remaining cleanup provisions are not conditional but consist simply of deleting the memory context and releasing the device context.

Painting from the Clipboard

In response to the `WM_PAINT` message, the *Capture* demo displays any bitmap image contained by the clipboard, regardless of the source of the image. Again, the first step (after initializing the customary device context) is to open the clipboard. This time, however, we do not call the `EmptyClipboard` function, and the application does not assume ownership, only access.

```
case WM_PAINT:
    hdc = BeginPaint( hwnd, &ps );
    OpenClipboard( hwnd );
    if( hBitmap = GetClipboardData( CF_BITMAP ) )
    {
        SetCursor( LoadCursor( NULL, IDC_WAIT ) );
        hdcMem = CreateCompatibleDC( hdc );
        SelectObject( hdcMem, hBitmap );
        GetObject( hBitmap, sizeof( BITMAP ),
            (LPSTR) &bm );
    }
```

This time, we use the `hBitmap` handle to retrieve the bitmap from the clipboard. If there is no bitmap image, the process is aborted.

Two methods of displaying a retrieved image are used: either actual-size or sized-to-fit. In the first case, we use the `BitBlt` function to execute a direct copy to the window. In the second instance, we call the `StretchBlt` function to copy the bitmap to fit the application window.

```
if( fExpand
{
    SetStretchBltMode( hdc, iStrMode );
    StretchBlt( hdc, 0, 0, cxWnd, cyWnd,
                hdcMem, 0, 0,
                bm.bmWidth, bm.bmHeight,
                SRCCOPY );
}
else
    BitBlt( hdc, 0, 0, cxWnd, cyWnd,
            hdcMem, 0, 0, SRCCOPY );
SetCursor( LoadCursor( NULL, IDC_ARROW ) );
DeleteDC( hdcMem );
}
```

Last, we must call the `CloseClipboard` function before the painting operation concludes:

```
CloseClipboard();
EndPaint( hwnd, &ps );
break;
```

As you can see, the big advantage of using the clipboard is simplicity. The operations involved are brief and uncomplicated, and their execution is quite speedy. Unfortunately, the corresponding operations directed toward building a bitmap file are not quite so simple and execute more slowly, although only marginally so.

File-Capture Operations

Capturing a bitmap to the clipboard is a task well-supported by Windows API functions, making this task almost automatic. In contrast, a similar capture to a file format is less well supported, requiring several specific subtasks to create a .BMP file image.

NOTE

The MFC `CBitmap` class does provide some further support. Unfortunately, this class implementation is woefully incomplete.

Creating the File Information Structures

A bitmap image file consists of three parts:

- A file header (BITMAPFILEHEADER)
- An information header (BITMAPINFOHEADER), which includes color-palette information
- The actual image information

Of these, the palette information and the image data vary in structure, depending on the type of color information and the encoding method (or lack thereof) used to store the image information.

Before any of these information structures can be created, however, the first step is to create a device context and to retrieve information about the device context and parameters, which will be used to describe the bitmap image. As with the process for capturing to the clipboard, this example begins by using the `HWND_DESKTOP` handle to retrieve a device-context handle. It then continues by querying the palette size and bits per pixel, as well as the horizontal and vertical device resolution.

```
int SaveBitmap()
{
    HDC      hdc, hdcMem;
    HANDLE   hBits, hFil;
    HBITMAP  hBitmap;
    HPALETTE hPal;
    LPVOID   lpBits;
    RGBQUAD  RGBQuad;
    DWORD    ImgSize, plSize, dwWritten;
    int      i, CRes, Height, Width, LnWidth, LnPad;
    BITMAPFILEHEADER bmFH;
    BITMAPINFO      bmInfo;
    LPLOGPALETTE    lp;

    SetCursor( LoadCursor( NULL, IDC_WAIT ) );
    //=== open file for write =====
    hFil = CreateFile( szFName, GENERIC_WRITE, 0, NULL,
                      CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL );
    if( hFil == NULL )
        return( ErrorMsg( "Can't open file" ) );
}
```



```

hdc = GetDC( HWND_DESKTOP );
CRes = GetDeviceCaps( hdc, SIZEPALETTE );
plSize = CRes * sizeof( RGBQUAD );      // palette size
bmInfo.bmiHeader.biBitCount =
    GetDeviceCaps( hdc, BITSPIXEL );
Height = GetDeviceCaps( hdc, VERTRES );
Width = GetDeviceCaps( hdc, HORZRES );
if( GetDeviceCaps( hdc, BITSPIXEL ) == 8 )
    LnWidth = Width;
else
    LnWidth = Width / 2;
if( LnWidth % sizeof(DWORD) )
    LnPad = sizeof(DWORD) - ( LnWidth % sizeof(DWORD) );
ImgSize = (DWORD)( (DWORD)( LnWidth + LnPad ) * 480 );

```

As the necessary raw information is retrieved, several local data variables are also calculated, including the raw image size, palette size, and the width of the individual scan lines.

The File Header The bitmap file begins with a file header defined by the `BITMAPFILEHEADER` structure, which holds information about the type, size, and layout of a DIB (device-independent bitmap) file.

NOTE

The terms bitmap and device-independent bitmap, along with the file extensions .BMP and .DIB, were once quite different. The .BMP extension identified only bitmaps using a device-specific structure supported by Windows 2.x. Today, device-dependent bitmaps are effectively obsolete. Common usage has allowed the .DIB extension to fall into disuse. The .BMP extension is used for all bitmaps, and it's assumed that all bitmap images are device-independent. Despite this assumption, however, the `BITMAPFILEHEADER` structure is still used.

The `BITMAPFILEHEADER` is defined as:

```

typedef struct tagBITMAPFILEHEADER
{
    WORD    bfType;
    DWORD   bfSize;
    WORD    bfReserved1;
    WORD    bfReserved2;
    WORD    bfOffBits;
} BITMAPFILEHEADER;

```

The `BITMAPFILEHEADER` fields are described in Table S15.1.

TABLE S15.1: BITMAPFILEHEADER Data Fields

Field	Description
bftype	Specifies the file type; must be BM
bfsz	Specifies the file size in DWORD units
bfReserved1	Reserved; must be zero
bfReserved2	Reserved; must be zero
bfOffBits	Offset in bytes from BITMAPFILEHEADER to the start of the actual bitmap in the file

The following code excerpt shows how these fields are set in the *Capture* demo:

```

bmFH.bftype      = 0x4D42;           // type is "BM"

bmFH.bfReserved1 = 0L;
bmFH.bfReserved2 = 0L;
bmFH.bfOffBits   = plSize +          // bitmap offset
                  sizeof( BITMAPINFO ) +
                  sizeof( BITMAPFILEHEADER );
bmFH.bfsz        = ImgSize +         // file size
                  bmFH.bfOffBits;
WriteFile( hFil, &bmFH, sizeof( bmFH ),
          &dwWritten, NULL );        // write file header

```

The `bfOffBits` field, which is the offset from the first of the image file to the beginning of the image data, is calculated as the palette size (`plSize`), plus the size of the `BITMAPINFO` structure, plus the size of `BITMAPFILEHEADER`. Once this file header is complete, this block of data is written directly to the file, using the `WriteFile` API function.

The Bitmap Information Structure Within the DIB file, the `BITMAPFILE-HEADER` structure is followed immediately by either a `BITMAPINFO` or `BITMAP-COREINFO` data structure. In the *Capture* demo, the `BITMAPINFO` structure defines the dimensions and color information for a DIB file. `BITMAPINFO` is defined as follows:

```

typedef struct tagBITMAPINFO
{
    BITMAPINFOHEADER  bmiHeader;
    RGBQUAD           bmiColors[1];
} BITMAPINFO;

```

The BITMAPINFO fields are described in Table S15.2.

TABLE S15.2: BITMAPINFO Data Fields

Field	Description
bmiHeader	BITMAPINFOHEADER containing information about the dimensions and color format of a DIB
bmiColors	An array of RGBQUAD data structures defining the colors in the bitmap

The Bitmap Information Header Structure The BITMAPINFOHEADER structure provides information about the size and organization of the bitmap image data and is defined as:

```
typedef struct tagBITMAPINFOHEADER
{
    DWORD    biSize;
    DWORD    biWidth;
    DWORD    biHeight;
    DWORD    biPlanes;
    DWORD    biBitCount;
    DWORD    biCompression;
    DWORD    biSizeImage;
    DWORD    biXPelsPerMeter;
    DWORD    biYPelsPerMeter;
    DWORD    biClrUsed;
    DWORD    biClrImportant;
} BITMAPINFOHEADER;
```

Table S15.3 describes the BITMAPINFOHEADER data fields.

TABLE S15.3: BITMAPINFOHEADER Data Fields

Field	Description
biSize	Number of bytes required by the BITMAPINFOHEADER structure
biWidth	Width of the bitmap in pixels
biHeight	Height of the bitmap in pixels
biPlanes	Color planes for target device; must be 1

Continued on next page

TABLE S15.3 CONTINUED: BITMAPINFOHEADER Data Fields

Field	Description
biBitCount	Bits per pixel; must be 1, 4, 8, or 24 (see Table S15.4)
biCompression	Type of compression for a compressed bitmap (see Table S15.5)
biSizeImage	Image size in bytes
biXPelsPerMeter	Horizontal resolution in pixels per meter of the optimum target device (applications may use this value to select from a resource group a bitmap that best matches the characteristics of the current device)
biYPelsPerMeter	Vertical resolution in pixels per meter of the optimum target device (applications may use this value to select from a resource group a bitmap that best matches the characteristics of the current device)
biClrUsed	Number of color indexes in the color table used by the bitmap (see Table S15.6)
biClrImportant	Number of color indexes considered important for displaying the bitmap; if 0, all are important

The **biBitCount** field of the **BITMAPINFOHEADER** structure determines the number of bits defining each pixel, as well as the maximum number of colors in the bitmap. This **biBitCount** field may be set to any of the values shown in Table S15.4.

TABLE S15.4: Bitmap Bit Count

Value	Description
1	Monochrome bitmap; bmiColors field must contain two entries and each bit in the bitmap array represents one pixel. If the bit is clear (0), the first color entry is used; if set (1), the second color entry is used.
4	Maximum 16 colors; bmiColors field must contain the maximum of 16 entries with each pixel in the bitmap represented by a 4-bit index to the color table.
8	Maximum 256 colors; bmiColors field contains a maximum of 256 entries with each pixel in the bitmap represented by a byte index to the color table.
24	Maximum 2^{24} colors; bmiColors field is NULL . Each pixel in the bitmap is represented by 3 bytes in the data array representing the relative pixel intensities of red, green, and blue.
32	Maximum 2^{32} colors; if bmiColors is BI_RGB , bmiColors field is NULL . Each pixel in the bitmap is represented by 3 bytes in the data array representing the relative pixel intensities of red, green, and blue. The high byte in each DWORD is ignored.

NOTE

In Windows, when **biCompression** is **BI_BITFIELDS**, only one 32 bits per pixel color mask is supported as blue = 0x000000FF, green = 0x0000FF00, red = 0x00FF0000. (The result, of course, is WHITE). In Windows NT, when **biCompression** is **BI_BITFIELDS**, bits set in each **DWORD** mask must be contiguous and should not overlap the bits of another mask. You do not need to use all the bits in the pixel.

The **biCompression** field of the **BITMAPINFOHEADER** structure identifies the compression format used, as listed in Table S15.5. (Bitmap compression formats are discussed later in the chapter.)

TABLE S15.5: Compression Format Identifiers

Field	Description
BI_RGB	Bitmap is not compressed.
BI_RLE8	Run-length encoded format for bitmaps, with 8 bits per pixel; uses a 2-byte format consisting of a count byte followed by a color-index byte.
BI_RLE4	Run-length encoded format for bitmaps with 4 bits per pixel; uses a 2-byte format consisting of a count byte followed by a byte containing two color indexes (nibbles).
BI_BITFIELDS	Bitmap is not compressed; color table consists of three DWORD color masks, which specify the red, green, and blue components, respectively, of each pixel. Valid when used with 16- and 32-bit-per-pixel bitmaps.

The **biClrUsed** field specifies the number of color indexes in the color table that are actually used by the bitmap. If the **biClrUsed** field is set to 0, the bitmap uses the maximum number of colors corresponding to the value of the **biBitCount** field, as listed in Table S15.6.

TABLE S15.6: Values for **biClrUsed**

Value	Description
0	Bitmap uses the maximum number of colors specified in the biBitCount field.
1..15	biClrUsed specifies the actual number of colors accessed by the device driver or graphics image.
16.. <i>nn</i>	biClrUsed specifies the size of the color table used to optimize performance for Windows color palettes. If biBitCount is 16 or 32, the optimal color palette starts immediately following the three DWORD color masks.

NOTE

If the bitmap is a packed bitmap—the bitmap array immediately follows the **BITMAPINFO** header and is referenced by a single pointer. The **biClrUsed** member must be either 0 or the actual size of the color table.

Colors in the **bmiColors** table should appear in order of importance, putting the highest frequency colors first. This way, if a bitmapped image is displayed on a device with a lower color resolution, the most important colors are mapped to the high-frequency colors in the palette.

In the *Capture* demo, the **BITMAPINFOHEADER** assignments are implemented as:

```
bmInfo.bmiHeader.biSize =  
    (DWORD) sizeof( BITMAPINFOHEADER );  
bmInfo.bmiHeader.biWidth      = Width;  
bmInfo.bmiHeader.biHeight     = Height;  
bmInfo.bmiHeader.biPlanes     = 1;  
bmInfo.bmiHeader.biCompression = BI_RGB;  
bmInfo.bmiHeader.biSizeImage   = 0L;  
bmInfo.bmiHeader.biXPelsPerMeter = 0L;  
bmInfo.bmiHeader.biYPelsPerMeter = 0L;  
bmInfo.bmiHeader.biClrUsed     = 0L;  
bmInfo.bmiHeader.biClrImportant = 0L;  
WriteFile( hFil, &bmInfo.bmiHeader,  
    sizeof( bmInfo.bmiHeader ),  
    &dwWritten, NULL );    // write info header
```

Writing the Bitmap Palette

Thus far, only the header information has been written to the bitmap file. Both palette and image information remain to be written. The *Capture* demo is set for two types of bitmaps: those with either 16- or 256-color palettes. (For monochrome, 24- or 32-bit-per-pixel bitmaps, additional provisions are necessary, as described earlier in this chapter.)

NOTE

32-bit-per-pixel color information is essentially the same as 24-bit-per-pixel data except for an 8-bit **NULL** in each entry used to pad the entry to a **DWORD** size.

Retrieving Palette Colors The following excerpt shows the handling for retrieving the palette color information and begins by allocating and locking sufficient memory space to contain the palette information:

```
// note: GHND = GMEM_FIXED | GMEM_ZEROINIT
hPal = GlobalAlloc( GHND, sizeof(LOGPALETTE) +
                  ( CRes * sizeof(PALETTEENTRY) ) );
// allocate memory for palette
lp = (LPLOGPALETTE) GlobalLock( hPal );
// lock the memory allocated
lp->palNumEntries = CRes;
lp->palVersion    = 0x0300;
// fill in size and version (3.0)
GetSystemPaletteEntries( hdc, 0, CRes,
                        lp->palPalEntry );
// and get the palette information
```

After allocating space for the palette information, the palette size is initialized (CRes) and the version number is set. With this done, the last step is calling `GetSystemPaletteEntries` to retrieve the actual palette color information and fill the `lp->palPalEntry` structure.

Converting Palette Colors Once we retrieve the palette information, but before the data is stored as part of the bitmap image, we must convert the `PALETTEENTRY` RGB order to the `RGBQUAD` format used by bitmap images.

The `PALETTEENTRY` structure is defined as:

```
typedef struct tagPALETTEENTRY
{
    BYTE  peRed;
    BYTE  peGreen;
    BYTE  peBlue;
    BYTE  peFlags;
} PALETTEENTRY;
```

In contrast, the `RGBQUAD` structure used by bitmap images is the same size—4 bytes—but uses an entirely different ordering for the colors. The `RGBQUAD` structure is defined as:

```
typedef struct tagRGBQUAD
{
    BYTE  rgbBlue;
    BYTE  rgbGreen;
    BYTE  rgbRed;
```

```
        BYTE   rgbReserved;
    } RGBQUAD;
```

As you can see, the PALETTEENTRY structure uses red-green-blue color order; the RGBQUAD structure uses blue-green-red. Therefore, provisions are necessary to convert the RGB order of the retrieved palette information to the bitmap's GRB order, as shown here:

```
    RGBQuad.rgbReserved = 0;
    for( i=0; i<=CRes; i++ )
    {
        RGBQuad.rgbRed   = lp->palPalEntry[i].peRed;
        RGBQuad.rgbGreen = lp->palPalEntry[i].peGreen;
        RGBQuad.rgbBlue  = lp->palPalEntry[i].peBlue;
        WriteFile( hFil, &RGBQuad, sizeof( RGBQuad ),
                   &dwWritten, NULL );
    }
```

As each palette entry is converted to RGBQUAD format, the converted entry is written to the bitmap file. Also, for each entry to the file, the rgbReserved field remains 0.

After looping through the palette information and creating the bitmap palette, the last step is to unlock and free the memory allocated to hold the palette.

```
    GlobalUnlock( hPal );    // don't forget to unlock
    GlobalFree( hPal );     // and release the memory
```

Still, the task is not finished. Thus far, the bitmap file and information header have been written, followed by the color palette data, but the image data has not been written yet.

Writing the Image Data

Earlier, I mentioned that various color resolutions use specific formats to encode the image data, even ignoring the data-compression formats entirely. As discussed, a 16-color image coded each pixel as a nibble of data (4 bits); a 256-color image requires a byte of data for each pixel; and a true-color (24-bit) image expects 3 bytes of data per pixel.

Rather than providing separate and different encoding methods for each color format, however, the CreateCompatibleBitmap function creates a memory device context that is compatible with the hardware device context (HWND_DESKTOP in the *Capture* demo). Then, by copying the bitmap image to this memory context, the bitmap bits are automatically rendered in the format appropriate to be written to the file.


```
hdcMem = CreateCompatibleDC( hdc );
hBitmap = CreateCompatibleBitmap( hdc, Width, 1 );
hBits = GlobalAlloc( GHND, LnWidth );
lpBits = (LPVOID) GlobalLock( hBits );
SelectObject( hdcMem, hBitmap );
```

Because a large bitmap requires substantial space (increasing with higher color resolutions), we use a small trick here. Instead of allocating memory space to contain the entire image at once, we allocate only enough space to contain one scan line from the image. And, this done, we lock the allocated space and select the bitmap into the memory context—the buffer, in effect.

The `SelectObject` function is only a setup for the device context. Although the bitmap has been selected to the context, no image data has yet been assigned to this bitmap. Therefore, in the next loop, the screen is read one scan line at a time, beginning at the bottom and working up, into the memory context; that is, into the bitmap that was sized for a single scan line.

```
for( i = Height - 1; i >= 0; i-- )
{
    BitBlt( hdcMem, 0, 0, Width, 1,
           hdc,    0, i, SRCCOPY );
    GetBitmapBits( hBitmap, Width, lpBits );
    WriteFile( hFil, lpBits, LnWidth, &dwWritten, NULL );
}
```

The whole purpose of this exercise, however, is not to create a bitmap one pixel high, but rather to use the `GetBitmapBits` function to copy this segment of image—first from the bitmap to the `lpBits` array and then from the `lpBits` array to the image file. In this fashion, instead of providing conversions to fit various color resolutions, the `BitBlt` function provides automatic conversion by writing the data to a memory bitmap. This memory bitmap is sized to match the original image and, therefore, is exactly the right size to be written to the file.

NOTE

It may have occurred to you that instead of using a loop, all of this could have been done in a single step (even though more memory would be required). What you must remember, however, is that the bitmap file also requires the image to be written from the bottom up. Copying the entire image in a single step would produce an inverted (mirrored) result. By using a loop, in addition to saving memory, you also avoid the necessity of inverting the image before writing the file.

Now, once the image has been written, the usual cleanup is necessary to release the various memory allocations and context and handle assignments:

```
GlobalUnlock( hBits );           // don't forget to unlock
GlobalFree( hBits );            // and release the image,
hBits = NULL;                   // optional but good form
DeleteDC( hdcMem );             // delete and release the
ReleaseDC( HWND_DESKTOP, hdc ); // device contexts
CloseHandle( hFil );            // and close the file
SetCursor( LoadCursor( NULL, IDC_ARROW ) );
return( TRUE );
}
```

Finally, the `CloseHandle` API function is called to close the completed bitmap image file, and the wait cursor is replaced by the default arrow cursor.

Bitmap Compression Formats

Compression is used with most image formats to reduce both memory and disk storage requirements. Windows supports two compression formats for bitmaps: one for 16-color images with 4 bits per pixel and one for 256-color images with 8 bits per pixel. The bitmap compression format flags are listed back in Table S15.5, and the `BI_RLE8` and `BI_RLE4` formats are described in the following sections.

The `BI_RLE8` Format

For 256-color images that use 8 bits per pixel to index pixels to the color palette, the `BI_RLE8` format offers two compression modes: encoded or absolute. Both of these modes may occur in the same bitmap (and usually do).

Encoded Mode Encoded mode uses `WORD` values, where the first byte in each `WORD` value specifies some number of consecutive values (01h..FFh) to be drawn using the color index indicated by the second byte. As an exception, the first byte may be set to zero to indicate an escape sequence, with the second byte denoting the end of a scan line, the end of the bitmap, or a delta escape, as listed in Table S15.7.

TABLE S 15.7: Compression Escape Sequences

First byte	Second byte	Definition
0	0	End of scan line
0	1	End of bitmap
0	2	Delta; the WORD value following the escape sequence contains horizontal (first byte) and vertical (second byte) offsets (relative) to the next pixel position

Absolute Mode Absolute mode is indicated by a WORD value, with the first byte set to zero and the second byte in the range 03h..FFh. The second byte represents the number of bytes following that contain absolute color indexes for a single pixel.

Since absolute mode also requires that each run be aligned on a WORD boundary, absolute runs are null-padded by byte as necessary to end each run on a WORD boundary.

BI_RLE8 Example Following is an example of hexadecimal WORD values from an 8-bit compressed bitmap, together with the corresponding decompression sequences:

0304 0506 0003 4556 6700 0278 0002 0501 0278 0000 091E 0001

Compressed Bytes	Decompressed Results / Pixel Values
03 04	04 04 04
05 06	06 06 06 06 06
00 03 45 56 67 00	45 56 67 (a single null byte is added for padding for this absolute mode run but is not included in the decompressed image)
02 78	78 78
00 02 05 01	Move 5 pixels right, 1 pixel down
02 78	78 78
00 00	End of scan line
09 1E	1E 1E 1E 1E 1E 1E 1E 1E
00 01	End of RLE bitmap

The BI_RLE4 Format

For 4-bit-per-pixel images, the BI_RLE4 format is used. Like BI_RLE8, this format incorporates two modes: encoded or absolute. Both modes may occur anywhere within an individual bitmap (and usually do).

Encoded Mode In the encoded mode, WORD values are used, and the first byte in each WORD value specifies some number of consecutive values (01h..FFh) to be drawn using the two color indexes contained in the second byte. Since the second byte contains two separate color indexes—one in the high-order nibble and one in the low-order nibble—the pixel sequence is drawn by alternating the two values indicated. With this sequence, the first pixel uses the first color index, the second pixel uses the second color index, the third pixel uses the first color index, and so on, until the indicated number of pixels have been written.

As an exception, the first byte may be set to zero to indicate an escape sequence with the second byte denoting the end of a scan line, the end of the bitmap, or a delta escape (see Table S15.7).

Absolute Mode In absolute mode, the first byte contains zero, while the second byte specifies the number of absolute color indexes (as nibble values) following. Subsequent bytes contain pairs of color indexes in the high- and low-order nibbles, with four color indexes in each WORD value.

Since absolute mode also requires that each run be aligned on a WORD boundary, absolute runs are null-padded by nibble as necessary to end each run on a word boundary.

BI_RLE4 Example Following is an example of hexadecimal WORD values from a 4-bit compressed bitmap, together with the corresponding decompressed sequences. (Single-digit values represent color indexes for single pixels.)

0540 0506 0005 4567 8000 0477 0002 0501 0478 0000 091E 0001

Compressed Bytes	Decompressed Results / Pixel Values
06 40	4 0 4 0 4 0
05 06	0 6 0 6 0
00 05 45 67 80 00	4 5 6 7 8 (three null nibbles are added for padding for this absolute mode run but are not included in the decompressed image)

Continued on next page

Compressed bytes	Decompressed Results / Pixel Values
04 77	7 7 7 7 7 7 7
00 02 05 01	Move 5 pixels right, 1 pixel down
04 78	7 8 7 8 7 8 7 8
00 00	End of scan line
09 1E	1 E 1 E 1 E 1 E 1
00 01	End of RLE bitmap

Windows 98 Graphics File Operations

Windows 3.x and earlier depended on the conventional C/C++ file operators to read and write files using, for example, the familiar `fopen/fwrite/fclose` functions. Windows 98 and Windows NT/2000 have replacements for these old standards in the form of API function calls, three of which are introduced here as direct replacements.

File-Open Operations

In any file operation, the first task is to open a file for either input or output. However, instead of the `fopen` function call familiar to C/C++ and Windows 3.1 programmers, you use the `CreateFile` API:

```
hFil = CreateFile( szFName, GENERIC_WRITE, 0, NULL,
                  CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL );
if( hFil == NULL )
    return( ErrorMsg( "Can't open file" ) );
```

Although the `CreateFile` function is equivalent to the `fopen` function, the calling format, as well as the options and capabilities supported, are different. The `CreateFile` function is defined as:

```
HANDLE CreateFile( LPCTSTR lpszName,
                  DWORD fdwAccess,
                  DWORD fdwShareMode,
```

```
LPSECURITY_ATTRIBUTES lpsa,  
                      DWORD fdwCreate,  
                      DWORD fdwAttrsAndFlags,  
                      HANDLE hTemplateFile )
```

The `CreateFile` function creates, opens, or truncates a file, returning a handle to the file for subsequent access.

If successful, the `CreateFile` function returns an open handle to the specified file. If the operation fails, the returned value is -1 (0xFFFFFFFF), and you can use the `GetLastError` function to retrieve extended error information.

NOTE

You can use the `CreateFile`, `WriteFile`, and `ReadFile` functions with named pipes and mailslots (with Windows NT) and with communication resources, although you may need to observe some special features or restrictions. For further details, refer to Chapter 6 in the book and to the online API function documentation on pipes.

The CreateFile Parameters

The calling parameters for the `CreateFile` API function are explained in the following sections.

lpszName This is the filename argument passed as a pointer to a null-terminated string. The `lpszName` parameter specifies the name of a file, pipe, communications resource, or console to be created or opened.

fdwAccess The `fdwAccess` parameter identifies the file-access type and may be either or both of the following flag values:

GENERIC_READ Provides file read access; permits data to be read from the file and the file pointer to be moved.

GENERIC_WRITE Provides file read/write access; permits data to be read from and written to the file and the file pointer to be moved.

fdwShareMode The `fdwShareMode` parameter specifies if and how the file can be shared and must be some combination of the following flag values:

0 File cannot be shared.

FILE_SHARE_READ File can be opened for read-only access by other applications; used to open the client end of a mailslot.

FILE_SHARE_WRITE File can be opened for read/write access by other applications.

lpSa The `lpSa` parameter is a pointer to a `SECURITY_ATTRIBUTES` data structure specifying file-security attributes. The file system, such as NTFS, must support security attributes before these have any effects.

fdwCreate The `fdwCreate` parameter specifies the action taken when the named file already exists or when no file with this name exists. This parameter must have one of the following values.

CREATE_NEW Creates a new file, failing if the specified filename already exists.

CREATE_ALWAYS Creates a new file, overwriting any existing file.

OPEN_EXISTING Opens an existing file but fails if no file exists.

OPEN_ALWAYS Opens an existing file or creates a new file if none exists.

TRUNCATE_EXISTING Opens an existing file, truncating the file to a zero size or failing if the named file does not exist. The file must be opened using `GENERIC_WRITE` access (see the `fdwAccess` parameter).

When `CreateFile` creates a new file, if the `fdwAttrsAndFlags` argument is not `NULL`, the file attributes and flags defined are ORed with the `FILE_ATTRIBUTE_ARCHIVE` bit (see Table S15.8 in the next section). In like fashion, if an `hTemplateFile` parameter (discussed later) is specified, `CreateFile` copies the extended attributes associated with the specified template file to the newly created file. Otherwise, the security attributes assigned to the new file are specified by the `lpSecurityAttributes` parameter. Last, the newly created file's length is set to zero.

When `CreateFile` is used to open an existing file, the `dwFlagsAndAttributes` and `hTemplateFile` arguments are simply ignored, as is the `lpSecurityDescriptor` member of the `lpSecurityAttributes` argument. However, the remaining flag values in the `SECURITY_ATTRIBUTES` structure remain valid.

fdwAttrsAndFlags The `fdwAttrsAndFlags` argument specifies the file attributes and flags assigned to a file. Any combination of the flags and attributes listed in Table S15.8 is acceptable, except that all other flag attributes override the `FILE_ATTRIBUTE_NORMAL` flag.

TABLE S15.8: File Attribute Flags

Attribute Flag	Meaning
<code>FILE_ATTRIBUTE_ARCHIVE</code>	Sets archive bit, marking the file for backup
<code>FILE_ATTRIBUTE_HIDDEN</code>	Marks the file as hidden; will not be included in an ordinary directory listing
<code>FILE_ATTRIBUTE_NORMAL</code>	Marks file with no other attribute bits set; valid only if used alone
<code>FILE_ATTRIBUTE_READONLY</code>	Marks file as read-only; cannot be written or deleted
<code>FILE_ATTRIBUTE_SYSTEM</code>	Marks file as part of or used exclusively by the operating system
<code>FILE_ATTRIBUTE_TEMPORARY</code>	Marks file as temporary; applications should write to this file only if absolutely necessary
<code>FILE_ATTRIBUTE_ATOMIC_WRITE</code>	Marks file as an atomic write file; applications should write to the file using atomic write semantics
<code>FILE_ATTRIBUTE_XACTION_WRITE</code>	Marks file as a transaction write file; applications should write to the file using transaction write semantics
<code>FILE_FLAG_WRITE_THROUGH</code>	Instructs system to always write through any intermediate cache and go directly to the file
<code>FILE_FLAG_OVERLAPPED</code>	Instructs system to initialize the file so that <code>ReadFile</code> , <code>WriteFile</code> , <code>ConnectNamedPipe</code> , and <code>TransactNamedPipe</code> operations, which take a significant time to process, will return <code>ERROR_IO_PENDING</code> ; this return may be used to implement flow control
<code>FILE_FLAG_NO_BUFFERING</code>	Opens file without intermediate buffering or caching by the system; all reads and writes are executed on sector boundaries, which is useful for rapid reads/writes of large data images
<code>FILE_FLAG_RANDOM_ACCESS</code>	Accesses file randomly (used by Win32 API to optimize file caching)
<code>FILE_FLAG_SEQUENTIAL_SCAN</code>	Accesses file sequentially from beginning to end; applications should not reposition the file pointer (used by Win32 API to optimize file caching)

Continued on next page

TABLE S15.8 (CONTINUED): File Attribute Flags

Attribute Flag	Meaning
FILE_FLAG_DELETE_ON_CLOSE	Instructs system to delete the file immediately after all file handles have been closed
FILE_FLAG_BACKUP_SEMANTICS	Marks file as being opened or created for a backup or restore operation
FILE_FLAG_POSIX_SEMANTICS	Accesses file according to POSIX rules; because POSIX rules allow multiple files with the same name, differing only in case, files created with this flag may not be accessible from DOS, Win16, or Win32 but may be accessed from WinNT

NOTE

When you use `FILE_FLAG_OVERLAPPED`, the system does not maintain the file pointer. Instead, the file position is passed as part of the `OVERLAPPED` structure argument to `ReadFile` and `WriteFile` calls. The `ReadFile` and `WriteFile` functions must also specify an `OVERLAPPED` structure. The `FILE_FLAG_OVERLAPPED` specification and `OVERLAPPED` structure permit separate processes or threads to execute simultaneous operations on a single file. Using the `OVERLAPPED` structure, each process is responsible for maintaining and updating its own file-position pointer.

hTemplateFile The `hTemplateFile` parameter specifies a handle with `GENERIC_READ` access to a template file, which supplies extended attributes for the file being created. Attributes derived from a template file override any attributes supplied as explicit parameters (by the `dwFlagsAndAttributes` and `lpSecurityAttributes` arguments).

File-Write Operations

The `WriteFile` function is defined as:

```

BOOL WriteFile( HANDLE hFile,
                CONST VOID *lpBuffer,
                DWORD nNumberOfBytesToWrite,
                LPWORD lpNumberOfBytesWritten,
                LPOVERLAPPED lpOverLapped )

```

Like the `fwrite` function that `WriteFile` replaces, the purpose of the function is to write data to a file, beginning at the position indicated by the file pointer. After the write is completed, the file pointer is adjusted by the number of bytes actually written, except when the file is opened with `FILE_FLAG_OVERLAPPED`. If the file handle was created for overlapped I/O, the application must explicitly adjust the position of the file pointer after the write.

The `WriteFile` function has the following parameters:

hFile Identifies the file to be written. The file handle must have been created with `GENERIC_WRITE` file access.

lpBuffer Points to a buffer containing the data to be written to the file.

nNumberOfBytesToWrite Specifies the number of bytes to be written to the file. A value of 0 is interpreted as a null write.

lpNumberOfBytesWritten Returns the number of bytes actually written to the file and is automatically zeroed before any work is done or any error checking is executed. This argument cannot be `NULL` and must be the address for a valid `DWORD` variable.

lpOverlapped Points to an `OVERLAPPED` structure, which is required if the file was opened as `FILE_FLAG_OVERLAPPED`. Otherwise, this argument may simply be passed as `NULL`. (See the “Overlapped File Operations” section.)

The `WriteFile` function does support a few features not normally encountered or not relevant during DOS file operations, two of which are applicable to conventional disk file operations:

- The `WriteFile` function will fail if the target file is locked by another process and the attempted write overlaps the locked portion.
- If `nNumberOfBytesToWrite` is zero, `WriteFile` does not truncate or extend the file. Instead, the `SetEndOfFile` function can be used to truncate or extend the file.

NOTE

Truncating a file is usually done to reset a file size to zero before rewriting the file with new data but is occasionally used to discard portions of a record file. Extending a file is commonly employed to add space to a file before executing a direct write. In most cases, using conventional file-management techniques, neither of these operations are necessary.

File-Read Operations

The `ReadFile` function replaces the traditional and familiar `fread` function. The `ReadFile` function is defined as:

```

BOOL ReadFile(
    HANDLE   hFile,           // file handle
    LPVOID   lpBuffer,        // address of input buffer
    DWORD    nNumberOfBytesToRead, // bytes to read
    LPDWORD  lpNumberOfBytesRead, // count of bytes read
    LPOVERLAPPED lpOverlapped ) // overlapped I/O structure

```

The `ReadFile` function reads data from a file, beginning at the position indicated by the file pointer. After the read is completed, the file pointer is adjusted by the number of bytes actually read, except when the file handle has been created with `FILE_FLAG_OVERLAPPED`. If the file handle was created for overlapped I/O, the application must adjust the position of the file handle after the read.

The following are the `ReadFile` function's calling parameters:

hFile Identifies the file to be read. The file handle must have been created using `GENERIC_READ` or `GENERIC_WRITE` file access.

lpBuffer Points to the buffer to receive data read from the file.

nNumberOfBytesToRead Specifies the number of bytes to read from the file.

lpNumberOfBytesRead Returns the number of bytes actually read and is automatically zeroed before any work is done or any error checking is executed. This argument cannot be `NULL` but must be a valid address for a `DWORD` variable.

lpOverlapped Pointer to an `OVERLAPPED` structure, which is required if the file was opened with `FILE_FLAG_OVERLAPPED`. Otherwise, this argument may be passed simply as `NULL`. (See the "Overlapped File Operations" section.)

If the return value is `TRUE` but the number of bytes read is reported as zero, the file pointer was beyond the current end of the file at the time of the read.

The `ReadFile` function will fail (returning `FALSE`) if part of the file has been locked by another process and the read overlaps the locked portion.

Overlapped-File Operations

The OVERLAPPED structure, which remains an optional argument in `ReadFile` calls when file sharing is not enabled, can be used to set a custom file pointer—that is, a custom pointer to a location (offset) within a file.

The OVERLAPPED structure is defined as:

```
typedef struct _OVERLAPPED
{
    DWORD   Internal;
    DWORD   InternalHigh;
    DWORD   Offset;
    DWORD   OffsetHigh;
    HANDLE  hEvent;
} OVERLAPPED;

typedef OVERLAPPED *LPOVERLAPPED;
```

The structure's fields are defined as follows:

Internal Reserved for system use; specifies a system-dependent status that is valid only when `GetOverlappedResult` returns without setting the extended error information to `ERROR_IO_PENDING`.

InternalHigh Reserved for system use; specifies the length transferred; valid only when `GetOverlappedResult` returns `TRUE`.

Offset `DWORD` value specifying the low-order 32-bits of the offset address for the transfer. The specification is a file position defined as a byte offset from the start of the file.

OffsetHigh Optional `DWORD` value specifying the high-order 32 bits of the offset address for the transfer.

hEvent Identifies an event to be set to the signaled state when the transfer is complete. The `hEvent` argument is optional; however, if an event is used, it must be identified here before calling the `ReadFile`, `WriteFile`, `ConnectNamedPipe`, or `TransactNamedPipe` API functions.

NOTE

Both the **Offset** and **OffsetHigh** fields are ignored when reading from and writing to named pipes and communications devices.

Most applications, when opening a file for read or write, will call `CreateFile` without using the `FILE_FLAG_OVERLAPPED` flag. In this case, both `ReadFile` and `WriteFile` calls can be made passing the `lpOverlapped` argument as `NULL`, initiating the read or write operation at the current file position. If, however, the `lpOverlapped` argument is provided, the read or write operation will be initiated at the file offset specified in the `OVERLAPPED` structure. In either case, neither `ReadFile` nor `WriteFile` will return until the file operation is completed.

Alternatively, if `CreateFile` was called using the `FILE_FLAG_OVERLAPPED` flag, the `lpOverlapped` argument is required to provide the current file position for both read and write operations. If this argument is not provided, both `ReadFile` and `WriteFile` will return `FALSE`, and `GetLastError` will report `ERROR_INVALID_PARAMETER`.

When a valid `lpOverlapped` argument is supplied, the read or write operation begins at the offset specified. However, either `ReadFile` or `WriteFile` may return before the operation is completed, returning a result of `FALSE`, and `GetLastError` reports `ERROR_IO_PENDING`. This provision allows the process calling `ReadFile` or `WriteFile` to continue with other tasks as the read or write operation continues independently.

File-Size Reports

The `GetFileSize` API function includes provisions to recognize and report on files that are larger than 4GB, even though such generous file sizes are currently unlikely.

```
DWORD GetFileSize
(
    HANDLE hFile,           // handle of file to get size of
    LPDWORD lpFileSizeHigh  // address of high-order DWORD for file size
);
```

The `GetFileSize` function returns a `DWORD` value reporting the low-order 32 bits of the file size. The second argument, `LPDWORD`, is an optional pointer to a second `DWORD` value, which will receive the high-order 32 bits of the file size.

If an error occurs, the return value in the low-order 32 bits will be `0xFFFFFFFF`. If the actual file size causes this same value to be returned, a call to the `GetLastError` function will report `NO_ERROR`. However, in general, even though a maximum file size of 1.8×10^{19} bytes can be reported, most present applications

can simply ignore the high-order 32-bit value (by using `NULL` as the calling argument) and assume that the actual file size is smaller than the 4GB still possible. For larger files, you would supply a pointer to a `DWORD` value to receive the high-order 32-bit value.

File-Close Operations

For file operations, the `CloseHandle` function replaces the familiar `fclose` function. The `CloseHandle` function requires only one argument: the file handle originally returned by the `CreateFile` function call.

`CloseHandle` invalidates the specified object handle, decrements the object's handle count, and performs object-retention checks. Once the last handle to an object is closed, the object is removed from the system. The `CloseHandle` function, however, does not close module objects.

NOTE

The `CloseHandle` function is not limited to closing files. It can also be used with handles for console input or output, events, file mapping, mutex, named pipes, processes, semaphores, and threads. See Chapters 5, 6, and 7 in the book for examples.

Image File Formats

Along with the Window's native BMP image format, many other image formats exist. You may need to work with various formats when you are importing and exporting images between applications.

Here, we will cover three popular image file formats: `.PCX`, `.GIF`, `.TIF`, and `.TGA`. Other formats that you may encounter include the GEM/IMG format, used by Ventura Publisher among others; the PIC or MacPaint format, used by the Apple Macintosh; and PostScript (`.EPS`) image formats.

Paintbrush's PCX Format

For a long time, ZSoft's Paintbrush (`.PCX`) format provided the de facto standard for non-Windows (DOS) bitmapped images. Most graphics programs contain some provision for conversion from their native formats to `.PCX` formats. Also,

the original PBRUSH.EXE paintbrush program distributed with Windows was written by ZSoft and included .PCX/.BMP conversion facilities.

As graphics devices have become increasingly sophisticated, the .PCX image format has kept pace. Instead of being a single format, the PCX standard comprises a series of formats including 8-, 16-, and 24-bit color formats, as well as true-gray and monochrome formats.

PCX File Structure

All PCX image files begin with a header defined as:

```
typedef struct tagPCXHEAD
{
    char    manufacturer;    // always 0xA0
    char    version;         // version number
    char    encoding;        // should be 1
    char    bits_per_pixel;   // color depth
    short   xmin, ymin;      // image origin
    short   xmax, ymax;      // image dimensions
    short   hres, vres;       // image resolution
    char    palette[48];     // color palette
    char    reserved;
    char    color_planes;     // color planes
    short   bytes_per_line;   // line buffer size
    short   palette_type;     // gray or color palette
    short   hscreen_size;     // horizontal screen size
    short   vscreen_size;     // vertical screen size
    char    filler[54];       // null filler
} PCXHEAD;
```

NOTE

For 16-bit Windows systems, the `xmin`, `xmax`, `ymin`, `ymax`, `hres`, `vres`, `bytes_per_line`, `palette_type`, `hscreen_size`, and `vscreen_size` fields have been commonly defined as `int`. For 32-bit Windows, because `int` is now defined as a 4-byte value, this definition has been changed to `short` to preserve the necessary 2-byte field length.

The header has the following fields:

manufacturer A check identifying the file as a Paintbrush format image and should always be 0x0A.

version Identifies the version of PC Paintbrush which created the image file. Valid values are 0 (no palette information; Paintbrush 2.5, the earliest incarnation), 2 (valid palette information), 3 (monochrome or the display's default palette), 4 (Paintbrush for Windows), or 5 (Paintbrush 3.0 or later, including 24-bit image files).

encoding Should always be 1, indicating that PCX's run-length encoding (RLE) has been used. Note, however, that other values may indicate newer versions and newer encoding schemes.

bits_per_pixel Reports the number of bits to represent each pixel (per color plane). Possible values are 1, 2, 4, 8, or 24.

xmin, ymin Specify an offset position for the upper-left corner of the image relative to the upper-left corner of the screen (or window). In most cases, no offset is specified, and `xmin` and `ymin` will be 0. (Even when an offset is specified, it is still the prerogative of the application, or programmer, to accept or ignore this offset as desired.)

xmax, ymax Specify the image dimensions. Note that the sizes indicated by `xmax` and `ymax` are off by 1, because the actual pixel count begins with zero. For example, for an image with a width of 480 pixels, the `xmax` value would be 479. The actual width and height of the image should always be calculated as:

```
width  = ( pcxHead.xmax - pcxHead.xmin ) + 1;  
height = ( pcxHead.ymax - pcxHead.ymin ) + 1;
```

hres, vres Provide the resolution of the device (or video mode) where the image was created; for most purposes, these values may be ignored.

palette A buffer that contains the palette color information for images with 16 or fewer colors (3 bytes per palette entry or 48 bytes in length). For larger palettes (such as 256-color palettes), this information is appended at the end of the image data. The palette structure used in either case consists of a series of RGB triplets, with the first byte in each defining the red level, the second defining the green level, and the third byte defining the blue level.

color_planes Defines the image's color planes. The value is 4 for EGA 16-bit color images or 1 for all other images, including monochrome.

bytes_per_line Reports the number of bytes to allocate for a scan-line plane. This value must be an even number and cannot be calculated by subtracting `xmin` from `xmax`.

palette_type Originates with the advent of VGA graphics systems with a value of 1 for gray-scale and a value of 2 for full color. This field is ignored in later versions of Paintbrush.

hsscreensize, vsscreensize Report the horizontal and vertical screen size (of the original system) in pixels. These fields were defined for Paintbrush IV and Paintbrush IV+; for all other versions, these should be NULL.

filler Pads out the header to 128 bytes and should be filled with nulls (zeros). (Future revisions may redefine the image header by using parts of the filler field for new purposes.)

The *ViewPCX* Demo: Reading a PCX Image



The *ViewPCX* demo demonstrates reading a 256-color PCX image and provides an example of a file lookup dialog box to select an image file. Note that no provisions are made for changing the filename extension or for displaying any format of PCX file except one with a 256-color palette.

Since all .PCX images use the same header structure, the first step for reading any .PCX file is to retrieve the header data. The *ViewPCX* demo uses the Windows `CreateFile` and `ReadFile` API functions. More important, *ViewPCX* makes use of the `OVERLAPPED` structure to maintain a custom file pointer while executing an asynchronous file read.

The first step is to open the file to read:

```
hFile = CreateFile( PCXFile, GENERIC_READ, 0, NULL,
                  OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL );
```

Of course, there is always the need for a provision to report possible errors. If the `CreateFile` function fails, the return value (in `hFile`) will be `INVALID_HANDLE_VALUE`, which is handled in a relatively typical fashion:

```
if( hFile == INVALID_HANDLE_VALUE )
{
    wsprintf( szBuff, "Error: %d - unable to open %s!",
              GetLastError(), PCXFile );
    ErrorMsg( szBuff );
    return( FALSE );
}
```

Assuming the file is opened correctly, the image header is retrieved:

```
ReadFile( hFile, &pcxHd, sizeof(PCXHEAD), &fRes, NULL );
if( ( fRes != sizeof(PCXHEAD) ) ||
    ( pcxHd.manufacturer != 0x0A ) )
```

```
{  
    CloseHandle( hFile );  
    ErrorMsg( "Not a valid .PCX file" );  
    return( FALSE );  
}
```

The `ReadFile` operation is followed by two checks: The first check tests `fRes` (the byte count actually returned) against the expected (and requested) byte size, ensuring that a complete header structure was found and retrieved. The second check tests the identification byte to ensure that it is identified as a PaintBrush PCX format image.

At this point, the image has been identified as, presumably, a proper PCX format. However, there are several possible PaintBrush formats; any further progress depends on the image type and the palette information (if any).

NOTE

It is easy to add decode and display provisions for black-and-white and 16-color palettes to the demo. For 24-bit color images, you will need to make slightly more elaborate provisions, beginning with a 24-bit video card and an appropriate driver. But, remember, no palette is included in 24-bit-per-pixel images because each pixel contains its own color information as a 3-byte RGB value.

Because 256-color palettes require 3 bytes per color, or a total of 728 bytes to define the palette, the .PCX file header lacks sufficient space to contain the palette information. Instead, the palette information is appended to the end of the PCX image file.

The logical first step, since we are assuming that this is a 256-color image, is to make sure that the image file is large enough to actually contain, at a minimum, the image header, plus a 768-byte palette, plus a 1-byte palette ID. This is easily accomplished:

```
fSize = GetFileSize( hFile, NULL );  
if( fSize < ( 769 + sizeof(PCXHEAD) ) )  
{  
    // wrong format - too small for palette  
    CloseHandle( hFile );  
    ErrorMsg( "Not a 256 color image format" );  
    return( FALSE );  
}
```

The next logical check is to test the header version number where a value of 5 indicates the presence of a palette. However, the mere presence of a palette of some size does not ensure that the image is a 256-color format. Therefore, the next step is to retrieve the palette information by using a seek to an offset from the end of the file.

Using conventional DOS file operations, this could have been done using the `fseek` function:

```
fseek( fp, -769L, SEEK_END );
```

However, using the `ReadFile` API function, a different approach is necessary to accomplish a similar task. In the *ViewPCX* demo, an initial offset value is assigned:

```
fPos.Offset = fSize - 769;    // seek palette start
fPos.OffsetHigh = 0L;
fPos.hEvent = NULL;
```

The assigned offset is the file size (`fSize`) minus 769, placing the file pointer one byte ahead of the expected palette. Since a file size greater than 4GB is not anticipated, the `fPos.OffsetHigh` field is set as zero and will be ignored. Last, the `fPos.hEvent` field is set as `NULL`, because no special reports or controls are needed.

After setting the offset, the `ReadFile` API is called to return a single byte that will be tested for the palette identifier. Then, after incrementing the offset to account for the byte just read, `ReadFile` is called a second time to retrieve the assumed palette information.

```
bResult = ReadFile( hFile, &chPal, 1, &fRes, &fPos ); // get palette ID
fPos.Offset++;                                     // advance pointer
bResult = ReadFile( hFile, &pcxPal, 768, &fRes, &fPos ); // get palette
```

Last, the version number, palette ID, and returned palette size are tested. If any of these three tests fail, the file is closed, an error message reports that the image was not acceptable, and the process exits:

```
if( ( pcxHd.version != 5 ) || // check version number
    ( chPal != 0x000C ) ||    // check palette ID
    ( fRes != 768 ) )        // check palette size
{
```

```

        CloseHandle( hFile );
        ErrorMsg( "Not a 256 color image format" );
        return( FALSE );
    }

```

Assuming that everything else is acceptable, the retrieved palette information is decoded to a format acceptable to the device context, a logical palette structure.

```

SetCursor( LoadCursor( NULL, IDC_WAIT ) );
//===== create palette =====
hPCXPal = GlobalAlloc( GHND, sizeof(LOGPALETTE) +
                    256 * sizeof(PALETTEENTRY ) );
lPal = (LPLOGPALETTE) GlobalLock( hPCXPal );
lPal->palVersion = 0x0300;
lPal->palNumEntries = 256;
for( i=j=0; i<256; ++i )
{
    lPal->palPalEntry[i].peRed   = pcxPal[j++];
    lPal->palPalEntry[i].peGreen = pcxPal[j++];
    lPal->palPalEntry[i].peBlue  = pcxPal[j++];
    lPal->palPalEntry[i].peFlags = PC_NOCOLLAPSE;
    // use PC_NOCOLLAPSE instead of PC_RESERVED — //
    // PC_RESERVED maps to nearest existing color //
    // but no good matches exist for this purpose //
}
hPCXPal = CreatePalette( lPal );
hOldPal = SelectPalette( hdc, hPCXPal, FALSE );
RealizePalette( hdc );           // palette is now active

```

After retrieving the image palette information, the application needs to return the image data. Using the DOS file functions, this task would have been accomplished as:

```

fseek( fp, 128L, SEEK_SET );

```

However, using the ReadFile API, the process reverts to using the OVERLAPPED structure to set the offset:

```

fPos.Offset = 128;           // set file ptr to image data

```

Because the PCX header, regardless of the image type, is always 128 bytes in length, finding the beginning of the image data is easy. However, decoding the data does require a few provisions.

Also, to simplify the decoding process, the image data is read and decoded one scan line at a time. However, because the image is RLE, it's hardly practical to determine exactly how many bytes are in a single scan line before reading the data. Therefore, in order to continually update the `fPos.Offset` value, a new variable, `Index`, is used during the decode process to determine how many bytes of data have been used and, before the next scan line is read, to increment the offset.

A 256-color PCX image always uses RLE, much the same as in other image formats. Thus, while reading the data, if the two high bits are set (the byte value is greater than `0xC0`), then the byte is read as byte count specifying the repeat count for the following byte. For example, the byte value `0xFE` indicates that the next byte read will be repeated `0x3E` or 62 times ($0xFE - 0xC0 = 0x3E$).

Now, as you may realize, this also means that individual pixels with palette values in the range `0xC0..0xFF`, which make up 75 percent of the total palette, require two bytes, rather than appearing as a single byte. Therefore, a run of pixels with the palette values `0xDE`, `0xDF`, `0xDF`, `0xEA`, `0xE2`, `0xE7` would be encoded as `0xC1`, `0xDE`, `0xC2`, `0xDF`, `0xC1`, `0xEA`, `0xC1`, `0xE2`, `0xC1`, `0xE7`, which is not precisely a savings. This does, however, illustrate the importance of building the palette with the high-frequency color entries appearing first. Still, shortcomings aside, RLE does generally reduce rather than increase file size.

There are still a few tricks involved in decoding an RLE image. For example, the decoding provision used in *ViewPCX* begins by initializing two values, `i` and `j`, before initiating a loop for the scan lines.

```
j = Index = 0;          // initialize position
while( j < depth )
{
    fPos.Offset += Index;
    i = Index = 0;
    ReadFile( hFile, &ImgArray, sizeof(ImgArray), &fRes, &fPos );
```

The offset field in `fPos` is incremented at the beginning of each cycle of the loop; the first time around, however, `Index` is already zero, so the offset remains at 128 bytes, which is the beginning of the image data. On subsequent cycles, after the offset is adjusted, `Index` is reset to zero in preparation for the following decode process. At this point, the `ReadFile` call has read a block of image data, beginning at the specified offset.

After reading the data, a new loop is initialized to handle decoding a single scan line from the image. Within this loop, the first step is a test to determine if the current byte is a repeat value:

```
do
{
    if( ( ImgArray[Index] & 0xC0 ) == 0xC0 )
    {
```

If the current byte is a repeat byte (greater than 0xC0), the count variable is derived, and a new for loop begins to paint the required number of pixels using the next byte in the data array as the palette index.

```
        count = ImgArray[Index++] & 0x3F;
        for( k=0; k<count; k++ )
        {
            SetPixelV( hdc, i++, j,
                      PALETTEINDEX( ImgArray[Index] ) );
            if( i >= pcxHd.bytes_per_line ) k = count;
        } // if line is too long just ignore any wraps
    }
```

The `SetPixelV` API function is essentially the same as the customary `SetPixel` API call, with one difference: Where `SetPixel` returns the existing color index for the pixel written, `SetPixelV` is marginally faster because it does not handle a return value. But remember that the image data is not an RGB value but rather a palette-index entry. However, both `SetPixel` and `SetPixelV` expect a `COLORREF` value, supplied here by calling the `PALETTEINDEX` macro with the palette index as an argument.

Last, purely as a precaution against encoding errors (which are not entirely unknown), if the repeat count extends beyond the scan-line length, the variable `k` is reset to terminate the inner loop.

If the initial test shows that the byte value is not a repeat count, `SetPixelV` is called once:

```
    else
        SetPixelV( hdc, i++, j,
                  PALETTEINDEX( ImgArray[Index] ) );
```

In either case, the `Index` value is incremented once more to point to the next element in the data array:

```
        Index++;  
    }  
    while( i < pcxHd.bytes_per_line );  
        j++;  
        i = 0;  
    }
```

The loop continues until `i` reaches the length of the scan line, after which, the vertical position (`j`) is incremented and the horizontal position (`i`) is reset to the first of the line.

Remember, the variable `Index` is used both to track the current position within the data array and, after decoding each scan line, to reset the `OVERLAPPED` offset before reading the next data block.

NOTE

The *ViewPCX* demo is included on the CD that accompanies this book, in the Supplement 15 folder.

Alternatives for Decoding PCX Images

Several alternatives are possible for decoding PCX images. The code shown in the *ViewPCX* demo is not the most efficient in terms of display; it was chosen to demonstrate using the `OVERLAPPED` structure.

One alternative is to use a different format to dynamically allocate an array large enough to hold all of the image data returned by a single `ReadFile` operation. With this method, a `DWORD` variable, such as `Index`, would keep track of the position within the data in the same fashion demonstrated within the loop in *ViewPCX*. The advantage, however, is that you need only one read operation, which would obviously accelerate operations.

A second alternative might be considered because the present method requires reading and decoding the image file every time a `WM_PAINT` message is received. You could change the handling, reading the image as a PCX file but creating a memory bitmap image from the result. This operation would be done only once, when the image file was selected rather than when the `WM_PAINT` message was received.

Then when a screen repaint is required, all that would be necessary would be to repaint the screen image using the `BitBlt` function, which is intrinsically faster than repeatedly reading and decoding the image file. Using the `SetPixelV` (or the `SetPixel`) function to paint a bitmap image is also intrinsically slow compared to using `BitBlt` (or `StretchBlt`) to simply transfer an image in bulk from a memory context to the screen context.

You can find the basics required for this second alternative in the `Capture-Bitmap` function in the *Capture* demo discussed earlier in this chapter (see the complete listing on the CD). Don't forget, however, to set the palette for the memory device as well as the active device context; otherwise, the resulting colors may be interesting but unexpected.

Implementing either of these alternatives, which you could also combine, is left as an exercise for the reader. However, the program listings (on the CD) include a code fragment showing how to create a bitmap from the PCX image data. As you will observe, the PCX image data must be decoded one line at a time before being transferred, using `BitBlt`, to build up the bitmap image.

24-Bit PCX Files

When you're working with 24-bit-per-pixel PCX images, remember that they do not contain any palette information. Instead, these images provide full 24-bit color information for each pixel in the image. Identified as version 5 or later, 24-bit PCX images store their data as 8 bits per color plane, in three planes. These are decoded in the same fashion as 16-color images, except that byte values (8 bits) are read as lines of red, green, and blue, in that order.

Therefore, to decode 24-bit PCX images, three scan lines are read as red, green, and blue image lines. After RLE decoding, these lines are treated by combining the first byte of each scan line as an RGB-triplet pixel value, rather than as a palette value. The second pixel uses the second byte from each scan line, and so forth.

Monochrome PCX Images

For monochrome PCX files, decoding is quite simple. First, if the two high bits of a byte are clear (ANDing with `0xC0`), then the six least-significant bits are written to the image as a series of six pixels. (If a bit is set, the pixel is on; otherwise, the pixel is off.)

If the two high bits are set, an index count is created by ANDing the byte with 0x3F and using this count (0..63) to repeat the next byte *count* number of times, optionally up to a total of 504 pixels (the repeated byte defines an 8-pixel series).

Obviously, the PCX encoding scheme is heavily weighted for use with images containing large contiguous areas. This is not, however, particularly efficient for scanned images (but, then, scanners were quite uncommon when the PCX format was created).

16-Color PCX Images

Paintbrush PCX images may have 2, 4, 8, or 16 colors before jumping to 256 color images. But, for 16 or fewer colors, the handling remains essentially the same, because the image is treated as four interleaved monochrome images, which is consistent with the EGA video format.

Although this format may sound mysterious, the reason lies in the structure of EGA video cards that were the intended environment for 16-color images. On EGA cards, four 32KB memory pages were treated as layers: one each for red, green, blue, and intensity. (Admittedly, this is an oversimplification.) The point is that the 4 bits selecting a palette color are written 1 bit to each plane, if you're working in machine language and accessing these planes directly.

In this circumstance, however, the question is how to decode the image, not the mechanics of an EGA card. To decode a 16-color PCX image, four scan lines are read and, initially, treated as monochrome masks. To create color (palette) information, the first bit from each scan line is combined after decoding, by shifting the bit from the second scan line one place left, the third scan line two places left, and the fourth scan line three places left to produce a 4-bit nibble.

This sequence of nibbles creates a single scan line for the image and can be written to the screen (or converted to another format) as index values to the 16-color palette.

CompuServe's Graphics Interchange Format

Perhaps one of the most popular image formats in general use—in terms of images available on bulletin boards, disk libraries, and CD-ROMs—is the Graphics Interchange Format (GIF). This format was developed by CompuServe as a vehicle for graphics images which could be transferred between different computer systems.

Although GIF is copyrighted by CompuServe, a blanket, nonexclusive, limited, royalty-free license has been granted to all developers, permitting free use of the GIF format in computer graphics applications.

The GIF format uses a very effective compression scheme, utilizing variable-length LZW compression (named for its developers: Lempel, Ziv, and Welsh). Although relatively complex to encode and decode, LZW compression has an important advantage over the simpler RLE compression schemes used by BMP and PCX images: reduced size. Images using LZW compression are virtually always considerably smaller than corresponding images created using RLE compression. LZW compression builds tables of patterns from the original, replacing repetitive patterns or pixel sequences with indexes to the table entries. LZW compression is also available in the public domain and is used in a variety of applications and forms, not just for image compression.

Other features supported by the GIF format include provisions for multiple images within a single file, local color tables including 256 colors, and interleaving scan lines (as used in PCX formats). The GIF format also has provisions for user-defined extensions.

GIF images are currently identified by two signatures, GIF87A and GIF89A, found in the first six bytes of the image and identifying, respectively, the original 1987 and 1989 revisions.

TIP

Current GIF standards and specifications (GIF89A) are readily available on CompuServe (GO GRAPHIC SUPPORT FORUM), as well as from a variety of other online services and private BBSs. A wide variety of GIF display programs, format conversion programs, source code examples, and GIF images are available through these same sources.

Tagged Image File Format

The Tagged Image File Format (TIFF) is perhaps the most complex of the popular formats. This format incorporates a variety of methods for describing images and, depending on the implementation, may provide several different means of data compression.

A second strength of the TIFF image format is that .TIF images, stored in uncompressed format, are capable of tremendous compression using standard file-compression utilities. Compressions of 97 to 99 percent are not uncommon.

This format is popular with typesetting and production graphics applications, partially because it was one of the earlier formats capable of supporting high-resolution images and partially because it provides several subformats optimized for different types of images. The following five classes of TIFF images are supported:

- Class B TIFF files consist of black-and-white images, coded as one bit per pixel and providing three compression formats: none, CCITT Group 3, and PackBits.
- Class G TIFF files are used for gray-scale images consisting of 4 or 8 bits per pixel (16 or 256 shades of gray) and are either uncompressed or use LZW compression.
- Class P TIFF files support color palettes using 1 to 8 bits per pixel and are either uncompressed or use LZW compression.
- Class R TIFF files are used for 24-bit-per-pixel images and, optionally, use LZW compression.
- Class F TIFF files are used for fax images.

TIFF formats can be used as demanded by circumstances. Remember, however, that these are provisions only of the TIFF specification. No actual implementation of the TIFF image software includes all possible formats or compression schemes. There are a number of other TIFF variations in use that do not follow any published standards. In general, these tend to consist of variant compression algorithms but may vary in other ways as well.

Typically, a TIFF encoder/decoder may run five to ten thousand lines of code.

TIP

The TIFF file specification and format instructions are available by request from either Aldus or Microsoft. Examples are available for a variety of systems.

Truevision's TARGA Format

The TARGA (TGA) file format, originally developed by Truevision, Inc., has been the predominate 24-bit image format used with frame-grabber boards. Truevision markets computer/camera interface boards that are used extensively on machine imaging, as well as for a variety of other applications. Using TARGA cards, or any of a variety of competing brands and models, images are captured directly from video cameras with pixel depths of 16, 24, or 32 bits.

Previously, a TARGA card (or equivalent) was required, usually along with a second, high-resolution monitor, to display TGA images. Today, you can use a single monitor and a wider variety of video cards that are capable of supporting 24-bit-per-pixel images, although these are not yet in widespread use.

All three of the image formats supported (16, 24, or 32 bits per pixel) can be considered true-color formats. And, speaking from personal experience in a color-critical application, I can report that the differences between images using these three formats are quite indistinguishable to the human eye, even tested in side-by-side displays of highly magnified gemstones.

Pixels in the 24-bit image format consist of three 8-bit color values in RGB order. The 32-bit format also contains three 8-bit color values, but also include a fourth 8-bit field, which is NULL and simply ignored, having no purpose except to pad the entry to a DWORD size for convenience in handling and alignment.

The third format, 16 bits per pixel, consists of three 5-bit color values with the sixteenth, high bit treated as an intensity bit. If the high bit is set, the three 5-bit image color values each correspond to the five most-significant bits in the corresponding 8-bit color values. If the high bit is cleared, the three 5-bit values are shifted right one 1 bit, decreasing color intensity.

TIP

TARGA image file specifications can be requested from Truevision, Inc, 7340 Shadeland Station, Indianapolis, IN 46256-3919. The phone numbers are 317-841-0332 (voice) and 317-576-7700 (fax).

Techniques for Converting 24-Bit Color Images

Although 24-bit true-color cards are becoming popular, they are still not common. For the present, SVGA (256-color) cards remain the high-resolution standard and are likely to continue so for at least the next few years. At the same time, the 24-bit video frame-capture systems are also popular but cannot be readily displayed on SVGA systems.

There is a solution: Convert 24-bit color images captured by video cameras to 256-color palette images, which can be displayed on most available systems. (Some graphics programs designed to manipulate 24-bit color images offer just such a palette-compression feature under the generic title Posterizing.)

Converting a potential palette of 16 million colors ($2^{24} = 16,777,216$) to a palette with a mere 256 colors does seem to be a considerable degradation in image quality, but it isn't really quite as bad as it sounds. While the potential palette size of a 24-bit image is over 16 million colors, the actual image (assuming 400×512 pixels) is a total of only 204,800 pixels. Assuming that no individual pixels share the same color, the reduction to a 256-color palette is only an 800-to-1 color reduction rather than 65536-to-1, an improvement of 82:1.

Commonly, however, a typical image will contain a much smaller range of actual colors—perhaps as many as 400 or 500 distinct shades, but usually fewer. And, even when the color variation is high, many shades that are technically different will still be relatively close in hue and can be represented by a single palette entry.

Several methods exist for converting 24-bit images to 256-color palette images. The simplest method, although not necessarily the best approach, is to begin by constructing a 256-color palette containing a range of hues that can be used for a variety of images. The drawback to this method is that the resulting palette does not match any image very well and the resulting displays have a rather cartoon-like quality about them. The following sections suggest some other methods.

A Frequency-Ordered Palette

A better method than constructing a simple 256-color palette is to begin by constructing a histogram of the colors represented in a specific image. This entails

processing the entire image to construct a frequency record for each individual color in the image.

After this is completed, you can build a custom, frequency-ordered palette from the highest frequency colors with the remaining image pixels mapped to their nearest equivalents from the palette. (Hint: Reserve two of the 256 palette entries: one each for pure white and pure black.)

Next, after constructing a palette of the high-frequency colors, map the original palette values as indexes to their corresponding palette entries or, if no matching palette entries exist, to the closest available palette entry.

A Distributed Palette

Another conversion approach follows the same general pattern of creating a histogram of the actual colors; however, instead of simply taking the 256 (or 254) highest frequency colors to create the palette, you create a distributed palette.

In this format, after creating a binary tree of color frequencies, the color tree is scanned for the total number of entries and for the range of differences between colors. If fewer than 256 entries are found in the tree, the palette can be constructed directly, based on frequency.

If more than 256 entries are found in the tree, you need to apply a color-conversion algorithm to find the closest matches in the tree, reducing the branches of the tree by eliminating the lowest frequency entries that have close matches. When the tree is reduced to a suitable number of entries, the remaining entries are used to create the palette.

If the number of branches (total colors) is, arbitrarily, less than one and a half times the palette size, a distributed palette is probably not necessary. On the other hand, if the number of colors exceeds this arbitrary threshold, a distributed palette may provide a better color spread than a frequency-ordered palette.

There are two considerations in selecting entries for a distributed palette: the uniqueness of the palette entry and the frequency of the color.

Taking the second consideration first, it should be fairly obvious that there's little benefit in devoting a limited resource—a palette entry—to a color that is used by very few pixels in the image. Precisely where this cutoff should be established is arbitrary, but in an image composed of 200,000 pixels, a frequency of 20 pixels is 0.01 percent of the total or 0.25 percent of the average. This value is low enough

to suggest that the color in question could be safely mapped to an existing palette entry.

The first consideration, uniqueness of a palette entry, is a different matter. This factor must be calculated carefully, taking all three of the color components (red, green, and blue) into account. The obvious method of comparing two color values is to sum the absolute difference of the red, green, and blue components:

$$dC = \text{abs}(R_1 - R_2) + \text{abs}(G_1 - G_2) + \text{abs}(B_1 - B_2)$$

The objective here, however, is to emphasize the difference between two colors and to find which colors in the image are the closest to each other and can, therefore, be represented by a single palette entry. The color difference (dC) can be better emphasized using a nonlinear formula:

$$dC = (R_1 - R_2)^2 + (G_1 - G_2)^2 + (B_1 - B_2)^2$$

This second formula shifts the weighting to emphasize differences in a single color component over difference distributed throughout the three color components.

For example, assume three colors, C1, C2, and C3, with RGB values 0x1F2C3B, 0x1F2A3B, and 0x1E2D3A, respectively. Using the first, unweighted formula, C1 and C2 have a color difference of 2 (in the green component); C1 and C3 have a color difference of 3 (1 each in the red, green, and blue values).

Using the second, weighted formula, however, the C1 and C2 entries have a weighted difference of 4 against the weighted difference of 3 for C1 and C3.

Still, as discussed previously when speaking of gray-scaled palettes and converting colors to true-grays, the human eye's response to colors is itself nonlinear. This weighting can be incorporated into a third formula, which calculates the difference between colors using the same weighting as the eye's response:

$$\begin{aligned} dC = & \text{abs}((R_1 - R_2) * 0.30) + \\ & \text{abs}((G_1 - G_2) * 0.59) + \\ & \text{abs}((B_1 - B_2) * 0.11) \end{aligned}$$

Using this third formula, the difference between the C1 and C2 color entries becomes 1.18 versus a difference between C1 and C3 of 1.00. This is a more appropriate result than the first formula yielded, but less distinctive than the second.

However, we can combine the second and third formulas:

$$dC = ((R_1 - R_2) * 0.30)^2 + ((G_1 - G_2) * 0.59)^2 + ((B_1 - B_2) * 0.11)^2$$

The difference between C1 and C2 becomes 1.39 versus a difference between C1 and C3 of 0.45.

This final revision of the weighting formula offers a distinctive difference in results both by incorporating the nonlinear response of the human eye and by emphasizing the difference in one color component over differences spread across all color components.

TIP

A small difference in speed could be achieved in the calculations by converting the percentage weights to integer values (for example, changing 0.30 to 30), thus removing all floating-point operations in favor of integer calculations. In most cases, however, this is not likely to provide a notable change in calculation times.

Experimenting with Color Differences

If you are interested, you can experiment with the *Color3* demo discussed in Chapter 24 to compare color differences. Here are some points to look for:

- What is the minimum total difference in all color components that can be readily identified by the human eye?
- What is the minimum difference in any one color component that can be readily identified?
- How do differences in each of the three component fields (red, green, and blue) compare?
- How do differences in intensity compare at different absolute intensities (how do absolute differences appear in proportion to absolute intensities)?

Ignoring extreme variations (commonly referred to as color blindness), color perception still varies widely between individuals and may also be affected by age, health, and the use of corrective lenses.

Graphics Selection Operations

- Area-selection tool features
- Adjustable target overlays
- Custom cursors
- Mouse-hit testing
- Bitmap file access and display

One aspect of graphics operations that is not commonly mentioned is how to select a section within an image or to select a region of interest. This requirement comes up quite frequently when working with live-video applications but is also applicable to static bitmaps.

In this chapter, we will look at a method of creating a nondestructive target overlay on top of an image. The sample program described here, *Target*, contains provisions for moving the target, resizing the target, and changing cursors to indicate which operations are being performed.

The *Target* demo also demonstrates a different approach to accessing bitmap files. It uses MFC classes and methods in place of some of the API functions and conventional operations illustrated in previous chapters.

Area-Selection Tools

A common requirement in many graphics operations involves selecting an area from either a static bitmap or an active video image. For both types of images, a nondestructive overlay works well as an area-selection tool.

Static Bitmap Area Selection

For static bitmaps, selection usually involves creating a tool to select an area, with the selection shown as an outline. For example, the Windows Paint program provides two selection tools: a free-form area tool and a rectangular area tool. Using the rectangular tool, you can select any rectangular region in an image then, subsequently, “pick up” or drag the selection. The free-form selection tool functions in the same fashion, except that you are allowed to “draw” an irregular region for selection.

The first case, rectangular selection, is the more common and is the type of selection discussed here. Selecting an irregular region involves much the same process, except that you must keep a list of boundary points and transfer the selected region as a series of image row sections.

The simplest way to select an area and to provide visual feedback to the user is to draw a rectangle enclosing the area on top of the existing image. Using a conventional drawing operation, however, is destructive to the existing image. Simply drawing a rectangle on top of a bitmap would be fine if you wanted to add

the rectangle to the image. But for selection, a different process is needed. You need to draw the rectangle using a method that allows the original image to be restored, without requiring redrawing the entire image.

The simplest method of drawing and then undrawing a figure is to use the ROP2 XOR operation, or R2_XORPEN, which is described in Supplement 12. Using the XOR drawing mode, the first time a shape is drawn, the drawing pen (and brush, if any) is XORed with the underlying image. This usually ensures that the drawn shape is optimally visible, regardless of the background image. More important, when the same shape is drawn a second time, the second drawing operation has the effect of canceling the first and, therefore, restoring the original background image, without needing to repaint the entire screen.

Aside from using the XOR drawing mode, the actual process of drawing the overlay is trivial. However, there is one caution: Whatever image or form is used for the overlay, it must be redrawn exactly to erase it before any changes occur in the position or size.

Active Video Image Selection

In the case of active video images, depending on the type of graphics capture card and processor, the selection process may involve capturing a static image first and then manipulating the static bitmap in much the same fashion demonstrated in the *Target* demo discussed in this chapter.

In other cases, where multiple video planes are supported, the selection process may be accomplished by drawing the area, or other targeting information, in a separate video plane and letting the system combine the targeting information with the active video for presentation. In the case of multiple video planes, drawing the overlay using the XOR mode still remains the fastest method of repeatedly drawing and removing targeting, selection, or region outline information.

WARNING

No matter how fast the system and the video card, restoring the entire image is still time-consuming and almost always unacceptable, especially when you're working with an active video image.

Area-Selection Tool Conventions

In many drawing applications, the current convention for area selection is to draw an overlay consisting of a dotted outline with rectangular handles at the

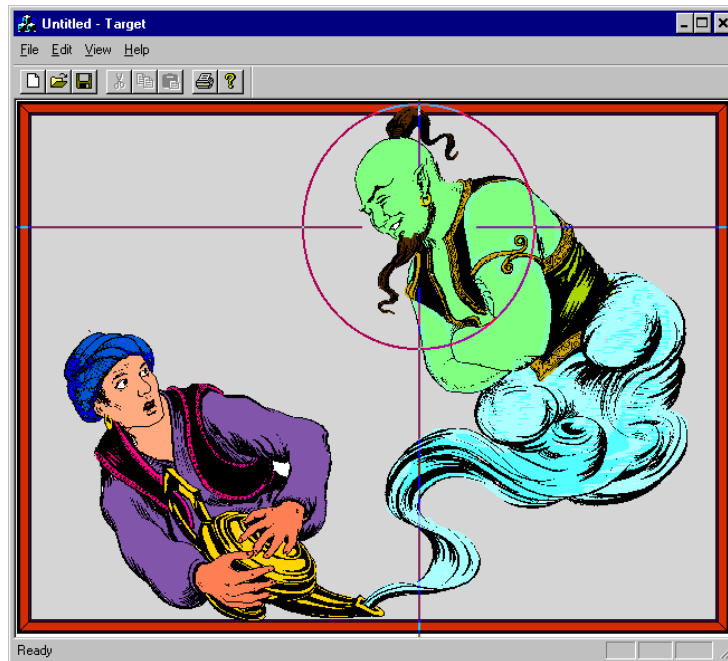
corners and centers of the sides. By placing the mouse cursor anywhere within the outline and pressing the mouse button, the selected region can be dragged to another position. On the other hand, by clicking on one of the handles, the outline can be dragged to a new size.

In the case of irregular areas, depending on the application, small “handles” may appear at nodes representing the vertices of a polygon outline. These handles are treated in the same fashion as a rectangular outline, permitting a vertex to be relocated. In other cases, such as in the Windows Paint program, no methods are provided for adjusting a free-form outline.

In the *Target* demo, a different set of conventions is used. For selection, you use a set of crosshairs that extend to the window margins and a circle that approximates the target area or region of interest (ROI). Figure S16.1 shows an example of a screen in the *Target* demo, with a bitmap displayed behind the target-selection overlay.

FIGURE S16.1:

Targeting an area in a bitmap



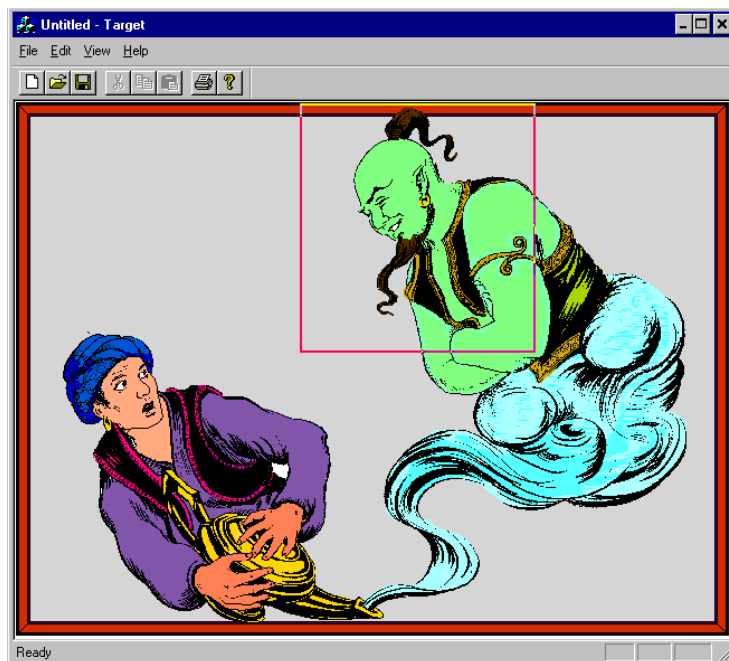
This format is common in machine-vision applications, where the user is selecting an area for examination. Because the crosshairs extend to the margins of the window, they can be used to indicate a position on scales along the sides. The center of the crosshairs is left open, so that the specific target is not obscured. The circular target-area marker is used to select an area for closer examination or for action by other associated tools. As an alternative, an elliptical, rather than circular, shape could be used for the target-area marker.

In most applications, after a user selects a rectangular or irregular region, the selection is moved, copied, or otherwise processed. During this processing, the common convention is to show an outline surrounding the selection. While it would be possible to capture or process a circular or irregular area, the usual practice is to process a rectangular image. This simplification is commonly used to show the actual area being processed, as well as to facilitate drag operations.

In the *Target* demo, when the right mouse button is pressed to initiate a capture (though no actual capture is done in this example), the crosshairs and circular target are replaced by a rectangle bounding but outside the region of interest. Figure S16.2 shows an example of the selection rectangle.

FIGURE S16.2:

Indicating the selected region of interest



Whether you use these conventions or any of several others depends on the needs of your application. There is no single set of conventions that apply to all cases and cover all requirements.

The *Target* Demo: Selecting Parts of an Image



The *Target* demo demonstrates using adjustable target overlays, testing for mouse hits with overlapping targets, and setting custom mouse cursors.

NOTE

The *Target* demo is included on the CD that accompanies this book, in the Supplement 16 folder.

Bitmap File Operations

The *Target* demo provides an option to open a bitmap file for a background image. It uses essentially the same bitmap file and display operations demonstrated in Supplement 15. However, there are a few differences because the version presented here relies heavily on MFC-defined classes rather than the standard APIs and conventional programming methods. You can look in the `CTargetView::ReadBitmap` procedure to see how these bitmap operations work.

Also worthy of your attention is the single `TRY...CATCH` exception handler used when opening a bitmap file:

```
TRY
{
    CFile cFile( csFName, CFile::modeRead | CFile::typeBinary );
    SetCursor( LoadCursor( NULL, IDC_WAIT ) );
    ...
    read the bitmap file here
    ...
}
CATCH( CFileException, e )
{
    #ifdef _DEBUG
```

```
        afxDump << "File access failed: " << e->m_cause << "\n";
    #endif
    return FALSE;
}
END_CATCH
```

The `CFile` constructor, which is used to open the file for reading, like any class constructor, does not return an error, regardless of what might go wrong. Therefore, to catch an error when opening a file in this fashion, the `TRY...CATCH` exception handling is required.

NOTE

C/C++ supports several forms of `try...catch` and `TRY...CATCH` exception handling. This latter form uses macros and is demonstrated in the *Target* demo. Refer to Chapter 9 in the book for more information about exception handling.

Mouse Responses

In the *Target* demo, the selection overlay needs to be able to respond to the mouse in several different fashions, depending on where the mouse is clicked:

- If the mouse is clicked in the center of the target, the target can be dragged to a new position but without changing the size.
- If the mouse is clicked on the left or right side of the target, or on the top or the bottom of the target, the target can be resized horizontally or vertically (but not both) without changing the center position.
- If the mouse is clicked on a corner of the target—upper left, upper right, lower left, or lower right—the target can be resized both horizontally and vertically, again without changing the center position.

In each case, the circular target is the focus of these operations; the crosshairs simply follow the center position of the target. Also, the background image remains unaffected by any of these operations.

Changing the Cursor

The *Target* demo also includes provisions to change the cursor to reflect the type of operation about to occur when the left mouse button is pressed. Although the

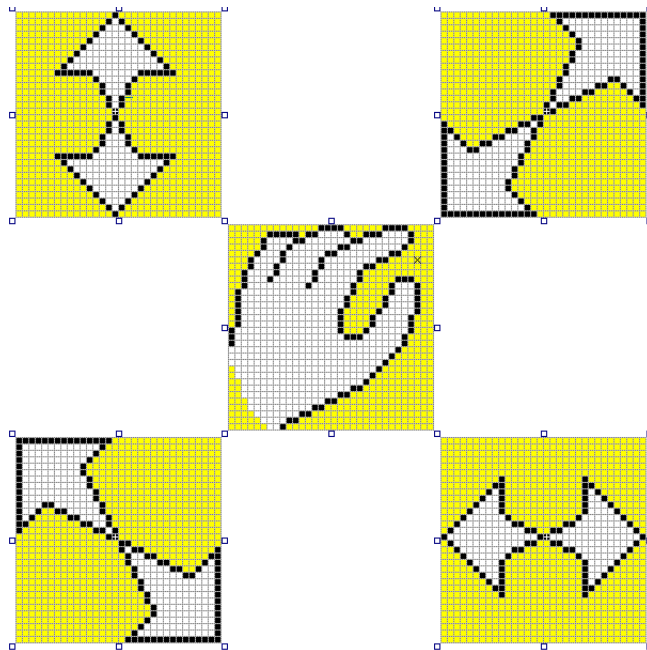
Windows GDI offers a variety of standard cursors, these are not always readily visible against a complex background (see Supplement 6 for more information about Windows cursors). The original program from which *Target* was derived provides a set of custom cursors, which have been incorporated into the demo as well.

These five custom cursors appear in Figure S16.3:

- North-south cursor (NS_CURSOR)
- Northeast-southwest cursor (NESW_CURSOR)
- Hand cursor (HAND_CURSOR)
- Northwest-southeast cursor (NWSE_CURSOR)
- East-west cursor (EW_CURSOR)

FIGURE S16.3:

Five custom cursors



In each case, the cursor image consists of a white body with a reversed outline. A small mark has been added to each image to identify the position of the cursor's hot spot.

Determining the Hit Position

To manage dragging and resizing operations for the target overlay, the first step is to determine where the mouse hit occurs; that is, the mouse's position when the primary mouse button was pressed.

In the `CTargetView` class, the `OnLButtonDown` method is called whenever the left (or primary) mouse button is pressed and receives two parameters: `nFlags` and `point`. In this case (as in most), we ignore the `nFlags` argument, which contains status information, and rely on the `point` information, which tells us where the mouse was when the event occurred relative to the application client window.

NOTE

By default, the mouse handler causes an `OnLButtonDown` call when the left mouse button is pressed. However, if the Mouse utility in the Control Panel has been used to swap the mouse buttons, the `OnLButtonDown` call responds to the right mouse button being pressed. For programming purposes, the primary mouse button always identifies itself with a `WM_LBUTTONDOWNxxxxx` message, and the secondary button always reports as `WM_RBUTTONDOWNxxxxx`, regardless of which physical mouse button is pressed.

Before any tests are made, the `m_nTrack` member is initialized as `NO_TARGET`. Subsequently, if a hit is identified in any target region, `m_nTrack` is reassigned a value to identify the correct region.

Also, before any other operations, the `SetCapture` method is called to ensure that mouse messages from outside the current window will still be received. The mouse capture will be released when the mouse button is released.

```
void CTargetView::OnLButtonDown(UINT nFlags, CPoint point)
{
    m_nTrack = NO_TARGET;
    SetCapture();
}
```

Next, the `m_xRadius` and `m_yRadius` members contain the size of the target ellipse and the `m_cPoint` member contains the centerpoint. To limit the drag operation to the center of the target area, the first target rectangle is defined using two-thirds of the vertical and horizontal radii. After creating the rectangle, the `NormalizeRect` function is called, purely as a precaution, to ensure that the bottom coordinate of the rectangle is greater than the top and the right side is greater than the left.

```

CPoint  cPoint( m_cPoint );
CRect   cRect( cPoint.x - ( ( m_xRadius / 3 ) * 2 ),
               cPoint.y - ( ( m_yRadius / 3 ) * 2 ),
               cPoint.x + ( ( m_xRadius / 3 ) * 2 ),
               cPoint.y + ( ( m_yRadius / 3 ) * 2 ) );

cRect.NormalizeRect();
if( cRect.PtInRect( point ) )

```

The `PtInRect` method simply returns `TRUE` if the `point` argument lies within the rectangle, or `FALSE` if not. While such a test is not difficult to perform, the provided member function is more convenient than writing a separate operation for each check that is made here.

If `PtInRect` returns `TRUE`, the `m_nTrack` member is set to `ALL`, meaning that the entire target overlay will be dragged when the next mouse-movement message is received, and the hand cursor is loaded as the active cursor.

```

{
    m_nTrack = ALL;
    m_hCursor = SetCursor( LoadCursor( theApp.m_hInstance,
                                       "HAND_CURSOR" ) );
}

```

If the `PtInRect` function returns `FALSE`, the `OnLButtonDown` method continues through a series of `else` statements, testing each target area in turn. If a hit is found, it sets the `m_nTrack` variable to the appropriate operation and loads the correct cursor.

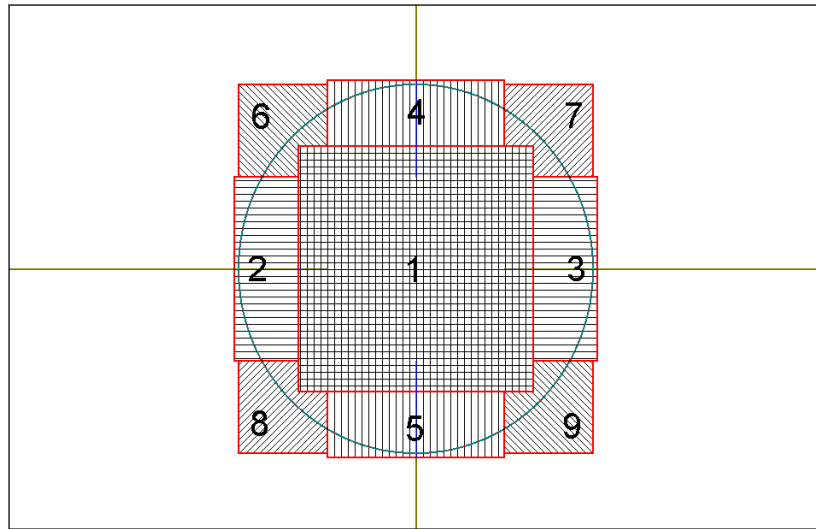
The real key here is to ensure that the target rectangles are tested in the correct order. Figure S16.4 shows nine overlapping target rectangles, numbered in the order tested.

What Figure S16.4 does not show is that region 1 overlaps regions 2, 3, 4, and 5. However, if a hit is found in the first region, no other regions are tested, making the overlapped areas irrelevant.

In like fashion, area 6 is overlapped by areas 1, 2, and 4. But because this area is tested last, a hit will be identified for this area only if it occurs within the irregular region shown. The same holds true for the regions identified as 7, 8, and 9; each is overlapped by three other regions that are tested first.

FIGURE S16.4:

The mouse-hit target rectangles



The point is that it's unnecessary to define complex hit areas when the same task can be accomplished by testing simpler regions in the proper order. On the other hand, when it is absolutely necessary to test complex regions, you can use other methods.

Last, the `CView::OnLButtonDown` method is called to provide default handling for the mouse messages.

```

    }
    m_bDrawOverlay = TRUE;
    CView::OnLButtonDown(nFlags, point);
}

```

Because we've already provided complete handling, calling the default method is optional, but it's still good practice.

Once we've decided where the mouse hit occurred, the next step is to wait for the `OnMouseMove` function to be called, indicating that the mouse has moved. The `OnMouseMove` method, like the `OnLButtonDown` method, is called with `nFlags` and `point` arguments, and again, the `nFlags` argument can simply be ignored as irrelevant.

Before doing anything based on the `m_nTrack` action flag, the next step is to decide if the mouse is still in the client window. If it is not—if the mouse has been moved outside the application window—then we will release the mouse capture and do nothing.

```
void CTargetView::OnMouseMove(UINT nFlags, CPoint point)
{
    CRect      cRect;

    if( m_bDrawOverlay )
    {
        GetClientRect( cRect );
        if( ! cRect.PtInRect( point ) ) // cursor outside of client area
        {
            m_nTrack = NO_TARGET;
            SetCursor( m_hCursor );
            ReleaseCapture();
            return;
        }
    }
}
```

Next, assuming the mouse is still in the window, the response is to call the `DrawOverlay` method to erase the existing target overlay.

```
DrawOverlay(); // erase the overlay target
switch( m_nTrack )
{
    case NO_TARGET: /* no action */
        break;
```

If the `m_nTrack` member indicates `NO_TARGET`, then we'll take no action. If `m_nTrack` is set to `ALL`, then the `m_cPoint` member needs to be updated as:

```
case ALL:
    m_cPoint.x = max( m_xRadius, min( point.x,
                                     ( cRect.right - m_xRadius - 2 ) ) );
    m_cPoint.y = max( m_yRadius, min( point.y,
                                     ( cRect.bottom - m_yRadius - 2 ) ) );
    break;
```

Alternatively, if `m_nTrack` is set to `TOP`, the vertical radius should be adjusted according to the mouse movement:

```
case TOP:
    m_yRadius = max( 30, m_cPoint.y - point.y );
    break;
```

The remaining case statements allow adjustments according to the quadrant selected, and the `switch` statement is followed by a test to ensure that the target

is not dragged outside the client window. The bulk of these provisions, however, are routine.

The one important provision remaining is to call `DrawOverlay` a second time to redraw the target overlay at the changed position or with the changed size.

```

        ...
        DrawOverlay();           // redraw the overlay target
    }
    CView::OnMouseMove(nFlags, point);
}

```

Finally, when the mouse button is released, the `OnLButtonUp` method is called. Here, the same arguments are supplied, but now both `nFlags` and `point` can be ignored. We don't really care where the mouse was released or what the flags were; our only interest is that the mouse button has been released. And the response to the mouse-button release is simple: Reset the member flags, restore the default cursor, and release the mouse-message capture.

```

void CTargetView::OnLButtonUp(UINT nFlags, CPoint point)
{
    m_bDrawOverlay = FALSE;
    m_nTrack = NO_TARGET;
    SetCursor( m_hCursor );
    ReleaseCapture();
    CView::OnLButtonUp(nFlags, point);
}

```

For the right mouse button, the provisions are even simpler: When the right mouse button is pressed, check and see if the event occurred in the target rectangle.

```

void CTargetView::OnRButtonDown(UINT nFlags, CPoint point)
{
    /*
    ///== routine to show target areas ===//
    DrawOverlay();
    DrawTargets();
    DrawOverlay();
    */

    CRect cRect( m_cPoint.x - ( ( m_xRadius / 3 ) * 2 ),
                 m_cPoint.y - ( ( m_yRadius / 3 ) * 2 ),
                 m_cPoint.x + ( ( m_xRadius / 3 ) * 2 ),
                 m_cPoint.y + ( ( m_yRadius / 3 ) * 2 ) );

    if( cRect.PtInRect( point ) )           // is cursor in client area?
    {

```

If the mouse event is in the target, set the capture flag, call `DrawOverlay` to remove the target overlay, and then call `DrawROITarget` to create the ROI outline.

```

        m_bCapture = TRUE;
        DrawOverlay();
        SetCapture();
        DrawROITarget();
    }
    CView::OnRButtonDown(nFlags, point);
}

```

If you were tracking mouse movement while the right mouse button is down, this would occur in the same `OnMouseMove` method used to track movement with the primary button down. The only difference would be that you would need to include some provisions, such as the `m_bDrawOverlay` and `m_bCapture` flags, to determine which type of event was being tracked. Or, more directly, the `nFlags` argument accompanying the mouse-movement message could be queried to find out which mouse button was pressed or if both buttons were pressed.

In this case, we really don't care about movement. All that we're waiting for is for the right mouse button to be released.

```

void CTargetView::OnRButtonUp(UINT nFlags, CPoint point)
{
    if( m_bCapture )
    {
        m_bCapture = ! m_bCapture;
        ReleaseCapture();
        Invalidate();
        DrawOverlay();
    }
    CView::OnRButtonUp(nFlags, point);
}

```

Once the right mouse button is released, we reset our flag, call the `Invalidate` function to redraw everything in the window, and then call `DrawOverlay` to restore the target overlay. The `DrawROITarget` function does not use `R2_XORPEN`; therefore, there is no easier way to remove the ROI target rectangle.

A Note about Custom Cursors

Some of you may have noticed that the use of the `SetCapture` method in the *Target* demo appears rather redundant since, as soon as the mouse moves outside the client window,

Continued on next page

`ReleaseCapture` has been called. Why call `SetCapture` and then release it as soon as it becomes useful?

The reason is that by calling `SetCapture`, you ensure that the cursor you assign to the mouse remains the active cursor and is not replaced by the default cursor as soon as the mouse moves.

The explanation for this behavior is found in the notes for the `SetCursor` function (from the MFC online documentation):

“If your application must set the cursor while it is in a window, make sure the class cursor for the specified window’s class is set to `NULL`. If the class cursor is not `NULL`, the system restores the class cursor each time the mouse is moved.”

The trick is how to set the class cursor to `NULL`. `SetCapture` provides a convenient alternative in this instance. However, the proper way to set custom cursors for a window under MFC is to intercept the `PreCreateWindow` function and to modify the `WNDCLASS` member of the `CREATESTRUCT` argument, setting the `hCursor` member to `NULL`. The revised `WNDCLASS` structure, however, must be registered before use through the `RegisterClass` function.

The long and the short of this is that setting custom cursors is not conveniently accomplished.

Other Methods of Interest

A provision has also been included in the source code to draw the several target areas used to test for mouse hits. This provision is found in the `DrawTargets` methods and was used to create the illustration shown previously in Figure S16.4. This provision can be enabled in the `OnRButtonDown` method.

Also of interest are the `OnFileOpen`, `ReadBitmap`, and `OnDraw` methods used in the *Target* demo. These parallel earlier examples but offer new versions using MFC classes and methods in place of some of the API functions and conventional operations illustrated in previous chapters.

The complete target-drawing and mouse-hit recognition operations are found in the `TargetView.H` and `.CPP` files, which are part of the *Target* demo included on the CD.

Graphics Printing Operations

- Procedures for copying images from a display context to a printer context
- Checks for a color or black-and-white printer
- Gray-scale definition
- Gray-scale printing enhancements
- Considerations for color printing

Being able to print a graphic image is almost as important as (or perhaps more important than) creating the image in the first place. The tools demonstrated in this chapter provide the basis for such facilities using both black-and-white and gray-scale color conversion.

If and how you use these features depends entirely on the needs of your applications. Most likely, you will need to adapt these features and perhaps also add some controls specific to your application. Alternatively, if you require only an occasional screen capture, you might prefer to combine the printer-output procedures with one of the screen-capture processes described in Supplement 16.

Incidentally, if you have access to color-reproduction facilities, you might also consider adding provisions for printing color separations; that is, printing separate red, green, and blue images for use as color screens in conventional printing processes.

Printer Operations

In many ways, Windows has greatly facilitated graphics image handling, with capabilities that range from providing hardware-independent graphics display environments to translating between different image formats. Just as Windows provides support for a wide variety of displays, it also provides support for a wide variety of printers, ranging from dot-matrix printers to all types of laser printers.

Furthermore, provisions exist (using the `BitBlt` or `StretchBlt` function) for copying images from a display context to a printer context, and from there to the printer itself. On the whole, Windows includes almost everything you need for hard-copy output of graphics images.

Unfortunately, almost is not everything. One fly remains in the ointment: Although most monitors are color, color printers are still less common than black-and-white printers. Windows does not offer any automatic solutions for printing color images to a black-and-white printer. As a general rule, when you direct a color image to a monochrome printer without any provisions for shading, the printer will print all colors except white as a solid black, which is generally not a very useful result.

Still, if no automatic solution has been provided, a custom solution is not beyond the realm of the possible and practical, as will be shown momentarily. But before tackling the solution, the first step is to understand the problem.

Win.INI versus Up-to-Date Printer Information

During installation, Windows offers an option to select one or more printers. When you choose printers to install, Windows copies the appropriate printer drivers to the Windows directory and lists these drivers in the registry.

NOTE

As explained in Chapter 8 of the book, the registry is a 32-bit Unicode data file, which you can access via the Registry Editor (**RegEdit.EXE** or **RegEdit32.EXE**).

Under Windows 3.1, the system stores the printer driver information in the Win.INI file, where a series of flag strings are used to locate and identify installed devices. The following is a fragment of a Win.INI file:

```
[windows]
...
device=HP LaserJet Series II,HPPCL,LPT1:
...
[devices]
HP LaserJet Series II=HPPCL,LPT1:
...
[HP LaserJet Series II,LPT1]
Paper Size=1
Number of Cartridges=1
...
[PrinterPorts]
HP LaserJet Series II=HPPCL,LPT1:,15,45
```

Under Windows 98, the old printer-control features continue to be implemented, even though the Win.INI file itself is obsolete. However, obsolete does not mean absent, and you probably have a Win.INI file in your Windows 98 directory. The problem is that any application that expects to find printer information in the Win.INI file may very well find that information, but the information may be completely out-of-date.

For example, in my own Windows 98 system, the Win.INI file identifies my system printer as an HP LaserJet, even though I installed a different printer after I switched to Windows 95, more than three years ago. On the other hand, the version of the Win.INI file in my Windows NT/2000 directory, which was not carried forward from an older Windows 3.x installation, does not contain any printer references.

The point is that applications written for Windows 2000 should always use the new printer-selection mechanisms and not rely on the old handling methods. Older methods are highly likely to access outdated or incorrect information.

The good news is that you do not need to write a printer-selection process for your applications. Instead, you can use the default printer-selection mechanisms supplied by Windows.

TIP

All applications created using Visual C++ and the AppWizard are supplied with a default File menu that contains Print, Print Preview, and Page Setup options, as well as default provisions to connect to the appropriate handlers and dialog boxes. The advantage is that this is all default code, which does not need to be duplicated. A common dialog box is provided for printer selection, including capabilities to connect to network printers. Users do not need to decipher a new selection mechanism.

Printer Queries

The Windows system and the MFC classes handle the task of getting and listing the available printers for you. However, certain applications may need to get other information about printer capabilities and limitations.

The *DC* demo, discussed in Chapter 12 of the book, demonstrates how to query the system device drivers and obtain information about device capabilities and limitations. In the *DC* demo's demonstration, all of the information available about a device is shown. Displaying all of these data elements requires a relatively long list of information requests. In other cases, instead of asking for everything available, you can make a more moderate request, restricting queries to only the appropriate or needed data.

The *GrayImage* Demo: Sending a Bitmap to a Printer



The *GrayImage* demo demonstrates both simple printer access and the gray-scaling of images. A number of the features in the *GrayImgView.CPP* section, such as selecting and displaying a bitmap image, should be familiar from demos discussed in earlier chapters, such as *Shades* and *ViewPCX*. The two

processes of interest in *GrayImage* are for drawing an image to the printer context and for converting an image from black and white to a printed, half-tone gray. For now, we'll begin with the procedures for sending an image to a printer.

NOTE

The *GrayImage* demo is included on the CD that accompanies this book, in the Supplement 17 folder.

The Printer Context

In previous examples, when a bitmap is presented in the client window, the bitmap image (file) is read, using the procedures demonstrated in those chapters. Then, when it's time to update (redraw) the client window, the bitmap palette is selected, a compatible memory device context is obtained, and the handle to the bitmap is used to copy the bitmap to the memory device context. The *GrayImage* demo uses essentially the same procedures to retrieve the bitmap:

```
void CGrayImageView::OnDraw(CDC* pDC)
{
    ...
    if( m_hBitmap )
    {
        if( m_pPal ) pOldPalette = pDC->SelectPalette( m_pPal, FALSE );
        nBitPxl = pDC->GetDeviceCaps( BITSPIXEL );
        nPlanes = pDC->GetDeviceCaps( PLANES );
        pDCMem = new CDC();
        pDCMem->CreateCompatibleDC( pDC );
        pDCMem->SelectObject( m_hBitmap );
```

In previous examples, once the bitmap is in the memory context, the `BitBlt` function is called to copy the image from the memory context to the display context—the client window.

```
pDC->BitBlt( 0, 0, m_bmWidth, m_bmHeight, pDCMem, 0, 0, SRCCOPY );
```

And, after copying the image to the display, a little bit of cleanup is performed to take care of the palette and the memory device context.

```
        if( m_pPal )
        {
            pDC->SelectPalette( pOldPalette, FALSE );
            pDC->RealizePalette();
        }
        delete pDCMem;
    }
```

Under previous versions of Windows, a separate procedure would have been required to copy a bitmap to a printer device. This print function would need provisions to query the installed printers, find out what the printer capabilities were, select a printer, get a printer device context, and finally write the image (or other data) to the printer queue.

Now, however, this task is greatly simplified because, when you select the printer icon from the toolbar or select Print from the File menu (assuming the application is being created with the AppWizard or another development tool), the `OnDraw` method is called with a pointer to a device context for output. For a printer-output operation, the only change from refreshing the screen is that the device context is a pointer to a printer context instead of a screen context.

Beyond this provision, the `OnDraw` function is expected to write to the printer in essentially the same fashion as it writes to the video display. The difference is that some of the output methods preferred for the video display may not be compatible with the printer context, even when writing to a color printer.

A Check for a Color or Monochrome Printer

Given the current popularity of color printers, not anticipating the presence of a color printer could be a serious error. If you simply attempt to provide gray-scaled output for hard-copy and the default color provisions for the video display, the result will be a blank sheet of paper when a color printer is encountered.

To understand why this happens, we need to take another look at the default process to copy the image to the screen, shown in the previous code fragment as:

```
//=== copy the bitmap to the screen ==(NORMAL)=====
pDCMem->SelectObject( m_hBitmap );
pDC->BitBlt( 0, 0, m_bmWidth, m_bmHeight, pDCMem, 0, 0, SRCCOPY );
```

Here, `pDCMem` is a memory context that is compatible with the device context supplied, `pDC`, when the `OnDraw` function is called by Windows. As long as `pDC` is a video context, `pDCMem` will be compatible with the `HBITMAP` object, `m_hBitmap`.

If `pDC` is a monochrome printer context, some degree of compatibility is still maintained. However, in most cases, if you use the `SelectObject` and `BitBlt` functions, the result will be that all colors (except white) are treated as black—not exactly the printout desired.

You can, however, persuade the monochrome printer to render approximate half-tones (gray-scale) by using the `SetDIBitsToDevice` function instead of the `SelectObject` and `BitBlt` functions. The only problem with this approach is that the gray-scale image produced will probably be rather coarse, which is the real reason for the gray-scale conversion routine discussed presently.

The next case occurs when the `SelectObject` and `BitBlt` functions attempt to print to a color context. The pDC context supplied by Windows for the printer drawing operation is normally a 32-bit per pixel scheme, which results in the `pDCMem->SelectObject` function failing, simply because the `pDCMem` context (created to be compatible with pDC) and the bitmap are not compatible. The result is a blank sheet of paper.

However, by using the `SetDIBitsToDevice` function and a pointer to the bitmap bits instead of a handle to a bitmap, you can avoid these potential problems. `SetDIBitsToDevice` can produce a color image on the screen, a native gray-scale image on a monochrome printer, and a color image on a color printer (leaving only the question of producing a better gray-scale image for our special routine). These rendering options and devices are summarized in Table S17.1.

TABLE S17.1: Color-Image Rendering Functions

Function/Device	SelectObject/BitBlt	SetDIBitsToDevice
Screen	Produces color image	Produces color image
Monochrome Printer	All colors except white treated as black	Colors rendered using native (printer) gray-scale
Color printer	Blank (no image)	Color image

So, why are we still using `SelectObject` and `BitBlt`? Simply because, when compatible with the desired operation, they are faster than the `SetDIBitsToDevice` function. Of course, for most video systems (printers are simply slower in any case) and in most cases, either route is sufficiently fast that the results will be indistinguishable, making the choice a moot point.

The revised `OnDraw` response begins, as before, by checking the bits-per-pixel and number of color planes to determine if you have a color or monochrome device.

```
if( ( nBitPx1 * nPlanes ) > 1 )
{
    if( m_pPal )
```

```
{
    pOldPalette = pDC->SelectPalette( m_pPal, FALSE );
    pDC->RealizePalette();
}
```

If this is a color device (either the screen or a printer), you want to select the palette for the image and realize it (make it active).

The next step is the critical decision. Since you don't know, without awkward tests, whether this is a printer context or a video screen, you can simply attempt the `SelectObject` operation.

```
if( pDCMem->SelectObject( m_hBitmap ) )
```

If `SelectObject` succeeds, you know that you have a compatible context and assume that this is the video device. Because the context is compatible (the bitmap was selected correctly), calling `BitBlt` will copy the image to the device.

```
pDC->BitBlt( 0, 0, m_bmWidth, m_bmHeight, pDCMem, 0, 0, SRCCOPY );
```

The alternative is that `SelectObject` failed, which probably means that this is a printer device context but still one supporting color. In this case, you want to call `SetDIBitsToDevice`, supplying the device-context handle (hDC) from the supplied device context, specifying the image size and position information, and providing pointers to the bitmap bits and to the bitmap information structure.

```
else
    SetDIBitsToDevice( pDC->m_hDC, 0, 0, m_bmWidth, m_bmHeight,
                      0, 0, 0, m_bmHeight,
                      m_pBits, m_pBmInfo, DIB_RGB_COLORS );
```

The final specification, `DIB_RGB_COLORS`, simply says how the color data is to be treated; it says that the bitmap information contains RGB colors in the color table. The alternative is `DIB_PAL_COLORS`, suggesting that the device palette should be used, which normally would not be appropriate.

NOTE

In many cases, especially when printing a hard-copy of an image, you may want to be able to resize the image. To resize the image on screen, the `StretchBlt` function is the ideal choice. But for the printer context, just as `SelectObject/BitBlt` fails, `SelectObject/StretchBlt` will also fail. Instead, for a printer device context, the choice would be to use the `StretchBlt` function to copy the image between memory contexts, resizing the image while doing so. Then, after resizing the image, use the `SetDIBitsToDevice` function to copy the resized image to the printer.

Last, to clean up, `SelectPalette` is called again to restore the original palette.

```
    if( m_pPal )
    {
        pDC->SelectPalette( pOldPalette, FALSE );
        pDC->RealizePalette();
    }
    else
        ... use gray-scale conversion to monochrome printer
```

Image Gray-Scaling

So what do you do when you need to output to a black-and-white printer but want something better than the relatively coarse default gray-scale supplied by most printer devices? The solution is to create a custom gray-scale by translating each color pixel in the original image to an array of black-and-white pixels. However, this also means that you must enlarge the original. If this is not convenient or practical, there remains the option of using the native gray-scaling.

Translating Colors to Gray Intensities

The first consideration in translating colors to grays is that, for the video display, colors are described by three digital values: red, green, and blue. Thus, using the `RGBTriplet` format, each of these color components has a value in the range 0 to 255 and the corresponding display ranges from black to full intensity. The result you see on the screen is a combination of the three primary colors. The relative intensities of each primary color, as well as the overall intensity, determine the “color” or hue perceived.

But the human eye is not linear. It responds differently to each of the three primary colors, with the strongest response (59 percent) to green. Our second strongest response is to red (30 percent), and our response to blue is the weakest (at 11 percent). Therefore, to translate red, green, and blue intensities into a gray scale, the absolute intensities of the three components must be weighted to match or, more accurately, the relative darkness of each component must be weighted to produce the appropriate portion of black ink on the page.

NOTE

For more details about translating colors to grays and creating gray-scale palettes, see Supplement 11, which covers Windows color handling and color palettes.

Defining Gray-Scale Patterns

Before matching colors to gray equivalents, it will help to have a range of grays for the matches. Thus, before writing the matching algorithm, the first step is to create a gray-scale for the printer. For demonstration purposes, we use a simple 16-step (4×4) gray-scale. However, for your own applications, you could implement a 25-step (5×5), 36-step (6×6), 64-step (8×8), or even 256-step (16×16) gray-scale.

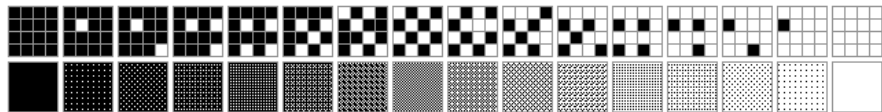
NOTE

The choice of a square gray pattern is dictated by convenience but is not quite an absolute. Using grays that are not squares, however, would require quite different handling and mapping and would produce distortion in the output image.

Figure S17.1 shows 16 4×4 matrices, ranging from full black to complete white. A sample of the resulting gray-scale appears below each 4×4 matrix.

FIGURE S17.1:

A gray-scale as a matrix series



WORD hex values providing binary descriptions of each pattern are 0x0000, 0x0400, 0x0401, 0x0501, 0x0505, 0x0525, 0xA425, 0xA5A5, 0xA7A5, 0xE5B5, 0xF5B6, 0xF5F5, 0xF5FD, 0xF7FF, 0xFFFF. As you may notice, one possible permutation of blacks and whites—0xF7FD (nine black to seven white, distributed among the 16 squares)—has been omitted.

These 16 patterns were selected to provide an even distribution and to avoid as much as possible any undesired elements, such as lines, herringbone patterns, or other artifacts.

Calculating a Gray Scale

As an alternative to creating a predefined gray scale, you can create a less rigorous (and somewhat more versatile) gray-scale by calculation. In its simplest form, each color pixel is mapped to a square grid in the printer context, as described in the previous section. However, instead of mapping the color pixel as a predefined pattern, each point in the square for the pixel is assigned a black or a white value in proportion to the calculated gray balance and the size of the square. In place of a predetermined pattern, you use a pseudo-random generator (such as C's random function) to assign the appropriate percentage of black pixels and white pixels in an essentially random pattern.

In general, this approach works best with a relatively large matrix for each pixel (8×8 or larger) and does not require an exact match between the range of grays used and the size of the pattern matrix. Of course, there are also a few disadvantages, such as a slight loss of edge definition, a need to keep the range of grays used relatively close to the matrix size, and some increase in mapping times because of increased complexity. But overall, the advantages can outweigh the disadvantages when wider ranges of grays are required.

Mapping Color Images to Gray Patterns

The process of mapping color images to gray patterns begins with a requirement for several new variables in the declaration, starting with `hTargetBM`, which is declared as an `HBITMAP` and serves as a buffer for the gray-scaled image during conversion. `pDCMem` and `pDCSrc` are handles for device contexts used while the color image is converted to a gray-scale image.

```
CDC      *pDCMem, *pDCSrc;
UINT     i, j, m, n;
int      nBitPxl, nPlanes, nGrayWd, nGrayHt;
BYTE     rVal, gVal, bVal, Gray;
DWORD    Color;
HBITMAP  hTargetBM;
WORD     Mask,
GrayPal[] =                // gray-scale masks
{ 0x0000, 0x0400, 0x0401, 0x0501,
  0x0505, 0x0525, 0xA425, 0xA5A5,
  0xA7A5, 0xE5B5, 0xF5B6, 0xF5F5,
  0xF5FD, 0xF7FD, 0xF7FF, 0xFFFF };;
```

The `GrayPal` (gray-palette) array (shown earlier in Figure S17.1) is declared here as an array of `WORD`.

Next, after we have decided that the device context supplied is a monochrome device, the `nGrayWd` and `nGrayHt` values are calculated.

```
else
{
    // if this is a monochrome device - i.e., a printer -
    //      get the max size supported by the device
    nGrayWd = min( (int)(4 * m_bmWidth), pDC->GetDeviceCaps( HORZRES ) );
    nGrayHt = min( (int)(4 * m_bmHeight), pDC->GetDeviceCaps( VERTRES ) );
    //      and create a gray-scaled bitmap to print
    hTargetBM = CreateBitmap( nGrayWd, nGrayHt, nPlanes, nBitPx1, NULL );
```

Our only restriction here is that we wish to ensure that the image we print is not larger than the output device supports. But we also need to make the output size 16 times larger than the original, providing space for a 4×4 gray pattern for each pixel in the original image.

Having calculated the necessary size, a temporary bitmap, `hTargetBM`, is defined with the necessary width and height and with the color planes (1) and bits per pixel (1) set for a monochrome image.

Next, if the bitmap creation fails, we simply abort the print operation with a minimal explanation. In your own applications, you would probably want to include a more informative explanation and some alternatives or suggestions for accommodations.

```
    if( ! hTargetBM )
    {
        ErrorMsg( "Bitmap creation error" );
        return;
    }
```

If the bitmap creation is successful, we proceed by calling the `SelectObject` function to select the (blank) target bitmap into the memory device context, which is compatible with the printer device.

```
    pDCMem->SelectObject( hTargetBM );
    // create a device context compatible with the
    //      (color) display context, not the printer context
    pDCSrc = new CDC();
    pDCSrc->CreateCompatibleDC( GetDC() );
    //      but select the gray-scaled bitmap to the context
    pDCSrc->SelectObject( m_hBitmap );
```

We also need a second, temporary device context, `pDCSrc`, which is compatible with the screen display. This context is provided by calling `CreateCompatibleDC` with `GetDC` as an argument to supply the display context. The original bitmap is selected here in the second device context, where we do not need to be concerned about compatibility.

At this point, we have a blank bitmap—four times wider and four times taller than the original—selected in the memory context, `pDCMem`, and the color bitmap selected in the temporary context, `pDCSrc`, which is also a memory device context. The actual conversion from color to gray-scale works between these two memory device contexts; that is, using `pDCSrc` as the source and writing the output to `pDCMem`.

```
for( i=0; i<m_bmHeight; i++ )
    for( j=0; j<m_bmWidth; j++ )
    {
        Color = pDCSrc->GetPixel( j, i );
        rVal = (unsigned char)( LOBYTE( HIWORD( Color ) ) );
        gVal = (unsigned char)( HIBYTE( LOWORD( Color ) ) );
        bVal = (unsigned char)( LOBYTE( LOWORD( Color ) ) );
```

Within a double loop (height and width), the color bitmap is scanned to determine R, G, and B values for each pixel. To convert these color values to a gray value, two algorithms are provided: a `TrueGray` algorithm and an unweighted conversion.

```
if( theApp.m_bTrueGray )
    Gray = (BYTE)( (UINT)(float)( ( rVal * 0.30 ) / 16 ) +
                    (UINT)(float)( ( gVal * 0.59 ) / 16 ) +
                    (UINT)(float)( ( bVal * 0.11 ) / 16 ) );
else
    Gray = (BYTE)( ( rVal + gVal + bVal ) / 48 );
```

The `TrueGray` algorithm produces a rather dark printed image; the unweighted algorithm results in a lighter printed image. For a screen display, the `TrueGray` algorithm offers the better match. For printed output, the unweighted one is preferable.

TIP

As an alternative, the application could use a logarithmic scale to keep blacks black and whites white but shift most colors toward the light end of the scale. You can try implementing this in the sample program if you're interested.

Once a color value has been converted to a gray intensity, the `DWORD Mask`, from the `GrayPal` array of predefined patterns, must be written to the output bitmap in `pDCMem`, as a 4×4 array, not as a linear string of bits. Remember that each pixel in the original is being written as a square of pixels in the output.

```
Mask = GrayPal[Gray];
//=== write gray mask to color bitmap context =====
for( m=0; m<4; m++ )
    for( n=0; n<4; n++ )
    {
        if( ( Mask >> ((m*4)+n) ) & 0x0001 )
            pDCMem->SetPixel( (j*4)+m, (i*4)+n, 0x00FFFFFF );
        else
            pDCMem->SetPixel( (j*4)+m, (i*4)+n, 0x00000000 );
    }
```

To write the output pixels, which are still written as full RGB values, the `Mask` value is tested bit-wise. The true bits are written as white, and the false bits are written as black.

And, finally, after the output bitmap is prepared, the `BitBlt` operation copies the image from the memory context, `pDCMem`, to the printer device context, `pDC`, where Windows assumes the rest of the task of handling the actual output.

```
    }
    //=== now copy from color context to printer context =====
    pDC->BitBlt( 0, 0, nGrayWd, nGrayHt, pDCMem, 0, 0, SRCCOPY );
    //=== the result is printed as black and white =====
    DeleteObject( hTargetBM );
    delete pDCSrc;
}
```

Once the `BitBlt` operation is handled, a minimum of cleanup is required. We only need to delete the target bitmap, `hTargetBM`, and the `pDCSrc` device context.

Overall, this may appear to be a rather roundabout fashion to map a color image to a gray-scaled equivalent. Still, this process does have several advantages, including these:

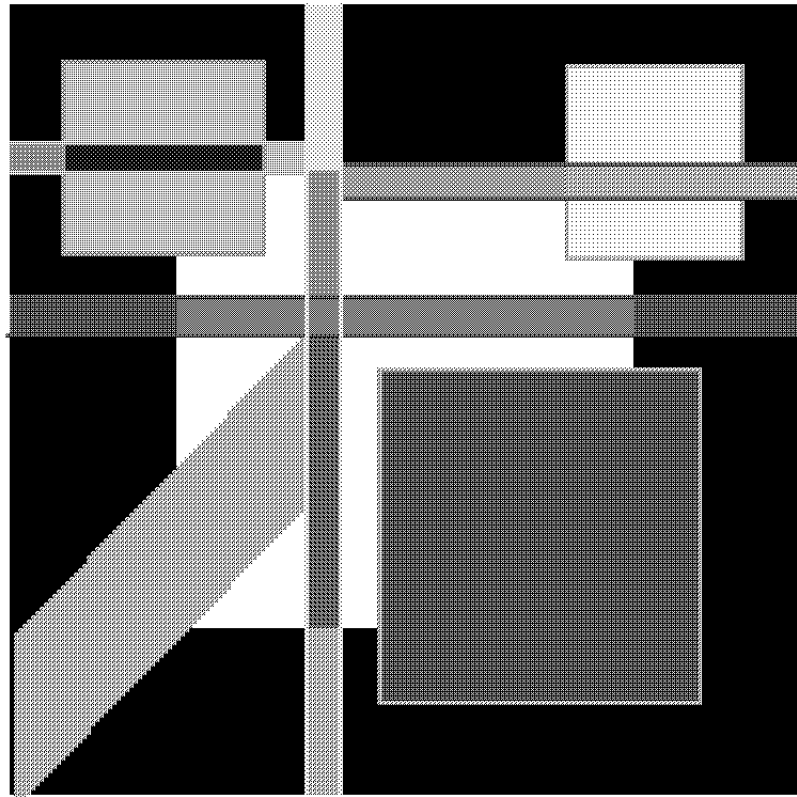
- There is no need for far long pointers to index bitmaps greater than 64KB.
- There is no need to convert palette color indexes into RGB values.
- On the whole, processing times are very fast.

Incidentally, as you may notice, the color-to-gray conversion itself tends to be considerably faster than the process of copying the gray image to the Print Manager.

Figure S17.2 shows an actual printout created using the process described here to convert the 256-color original Modern.BMP to a gray-scaled hard copy. The original output was executed on a 300-dpi laser printer.

FIGURE S17.2:

A true gray-scaled printout



Improved Gray-Scale Printing

In the *GrayImage* demo (in the *OnDraw* method), the *BitBlt* API is used to copy a gray image to a monochrome printer. However, the *BitBlt* API does not include any provisions for resizing the image. As an alternative, you can use the *StretchBlt* API to create any size output image desired, and you might use *CRect* coordinates derived from the printer context to size the bitmap to fit

the entire page. However, for several reasons, this may not always be an optimal choice. In actual graphics printing applications, you may want to improve the output in the following ways:

- Allow a specific size.
- Preserve the vertical and horizontal proportions.
- Avoid plaiding in the printed image.

Fortunately, all of these conditions are relatively easy to fulfill.

The first objective, a precise size, is simplicity itself. In the `StretchDIBits` operation, replace the page size with the desired image size. Just remember to convert from inches or millimeters (or whatever unit you're using) into logical-device coordinates (pixels or printer dots). For this purpose, device-resolution information is available using the `GetDeviceCaps` function (see Supplement 12).

The second objective, maintaining proportion, is equally easy. For the maximum image size, compare the horizontal and vertical size ratios, and then adjust the greater ratio to maintain image proportions.

Avoiding plaiding is perhaps the most difficult objective, simply because this provision is not completely compatible with either of the first two. Even so, in execution, it is not exceedingly difficult. In practice, the simplest solution is to size the image so that the dots in the output image are some multiple of the original pixel size (or, for gray-scales, some multiple of the gray-scaled pixel size). Thus, for a 200×200 pixel image converted into a 16-level gray-scale, the gray image is 800×800 pixels and could be printed as 800×800, 1600×1600, or 2400×2400 dots. Assuming a 300-dpi laser printer, the largest (2400-dot) image would be 8 inches wide.

NOTE

For dot-matrix printers with lower resolutions, the choices and possibilities are more restricted. Typesetting printers and many of the newer laser printer designs offer more versatility.

The only real problem is found with devices that lack a 1-to-1 horizontal-to-vertical aspect ratio. For example, some dot-matrix printers might provide a horizontal resolution of 96 dpi, while their vertical resolution is 180 dpi, for a ratio of 96-to-180 or 8-to-15. This is not an easy ratio to fit without distorting the image

proportions. Fortunately, most laser and ink-jet printers do have 1-to-1 aspect ratios.

Gray Images Printed in Color

In some cases, you might need to print a color hard-copy from a gray-scale original. For example, infrared photography images, particularly video images, are captured as gray-scale. In similar fashion, low-light (night-scope) images, NMR, and even CAT scan images do not have inherent color information but are gathered as images scaled by intensity, density, or a synthetic scale.

With infrared images, the problem is that we do not see in this portion of the spectrum, even if we did have printers or monitors capable of displaying these frequencies. But, at the same time, a black-and-white image is less informative than a color image.

In all of these cases, a common solution is to produce a false-color image in which colors are assigned (arbitrarily or otherwise) to various intensity ranges. How colors are assigned is subject to several considerations:

- What is the available range of information? How many gray, intensity, or other levels of information are available in the original image or data?
- How many colors can be displayed? Those who are working with more sophisticated forms of imaging equipment are rarely limited in their display capabilities and can usually assume at least a 256-color display capacity. However, printers may be more restricted in their color ranges.
- How large a range of color is actually needed? Do you need to use 256 colors or will a simpler palette of 16 colors serve just as well or even better?
- What color palette will best serve to display the information? In general, converting intensity to color is done to make certain characteristics stand out for easy identification and recognition.

As an example, one intensity-to-color development was done as a color printer driver for a company involved in infrared imaging. The image was captured using a special (and expensive) video camera and capture board and held a wide range of intensity data (at least 256 levels). The limiting factor was not the video display but the printer, which supported only a 16-color palette. Also, the printer palette needed

to accommodate a background color that was used for low-temperature areas in the image. The color chosen for the background was a light-blue palette entry for a neutral backgrounds. Black was used as a temperature threshold marker. Temperatures (intensities) below a certain level were mapped to violets, blues, and greens, advancing to black, and then to reds and yellows to show the higher intensities.

Color Images Printed in Color

In recent years, a variety of color printers have appeared on the market, ranging from paint-jet printers that produce medium-quality color images to dye-diffusion printers that produce more expensive but near-photographic-quality color images.

Unlike video displays, which use an RGB color scheme (as explained in Chapter 24), all printers use the complementary CYM (cyan-yellow-magenta) color scheme to print color. The complementary inks absorb everything except cyan, yellow, or magenta, respectively. But, by combining yellow and cyan, everything except green is absorbed, and the result is a green image. In like fashion, reds, blues, and all other shades are created by varying the combination and amount of each ink to leave only the desired color reflected. White, of course, is provided by the paper without ink; black uses all three inks to absorb all colors.

To the general relief of programmers, Windows supplies drivers for almost all printer types, including color printers. As for those printers that are not currently supported, most manufacturers are busy developing Windows drivers for them. Of course, there may be a few who are blithely attempting to ignore the new paradigm, but this may also be taken as a benchmark of their probable future and your own future expectations from such companies. Still, the usual cautions apply: Check available support before investing in a specialty printer, not after.

In general, printing a color image is quite similar to printing a monochrome image, as demonstrated in the *GrayImage* demo discussed in this chapter. The one difference is that no color palette is written to the output device context for monochrome images. For color printers, a palette must be supplied.

Before supplying a palette, you will need to ask the printer what its palette capacity is and supply a palette of the appropriate size. For limited palette sizes, you may need to construct an appropriate palette. Still, you can have a lot of fun

simply experimenting, or if you're too busy, turn a teenager (or pre-teen) loose on the problem and see what he or she comes up with.

The *GrayImage* demo included on the CD provides default handling for a color printer as well as gray-scaled output to a monochrome device.

Graphics Selection Operations

- Area selection tool features
- Adjustable target overlays
- Custom cursors
- Mouse-hit testing
- BMP (bitmap) file access and display

One aspect of graphics operations that is not commonly mentioned is how to select a section within an image or to select a region of interest. This requirement comes up quite frequently when working with live video applications but is also applicable to static bitmaps.

In this chapter, we will look at a method of creating a nondestructive target overlay on top of an image. The sample program described here, *Target*, contains provisions for moving the target, resizing the target, and changing cursors to indicate which operations are being performed.

Also, while we have previously used bitmap images in applications, the *Target* demo provides another example of accessing a bitmap file (look in the `CTargetView::ReadBitmap` procedure).

Creating a Selection Tool

A common requirement in many graphics operations involves selecting an area from either a static bitmap or an active video image. For static bitmaps, selection usually involves creating a tool to select an area, with the selection shown as an outline. For example, the Windows Paint program provides two selection tools, a free-form area tool and a rectangular area tool. Using the rectangular tool, you can select any rectangular region in an image, then subsequently “pick up” or drag the selection. The free-form selection tool functions in the same fashion, except that you are allowed to “draw” an irregular region for selection.

The first method, rectangular selection, is the more commonly used and is the type of selection discussed here. Selecting an irregular region involves much the same process, except for keeping a list of boundary points and transferring the selected region as a series of image row sections.

Drawing an Overlay

The simplest way to select an area and to provide visual feedback to the user is to draw a rectangle enclosing the area on top of the existing image. Using a conventional drawing operation, however, is destructive to the existing image. Simply drawing a rectangle on top of a bitmap would be fine if you wanted to add the rectangle to the image. But for selection, a different process is needed. You need

to draw the rectangle using a method that allows the original image to be restored, without requiring redrawing the entire image.

The simplest method of drawing and then undrawing a figure is to use the ROP2 XOR operation, or R2_XORPEN, which is described in Supplement 12 (on the CD accompanying this book). Using the XOR drawing mode, the first time a shape is drawn, the drawing pen (and brush, if any) is XORed with the underlying image. This usually ensures that the drawn shape is optimally visible, regardless of the background image. More important, when the same shape is drawn a second time, the second drawing operation has the effect of canceling the first, and therefore restoring the original background image, without needing to repaint the entire screen.

Aside from using the XOR drawing mode, the actual process of drawing the overlay is trivial. However, there is one caution: Whatever image or form is used for the overlay, it must be redrawn exactly to erase it before any changes occur in the position or size.

TIP

Incidentally, XOR drawing is also one of the simpler techniques used for animation. By performing an XOR draw operation to create an image and then a second XOR draw to erase the original (and restore the background) before relocating the image and beginning another cycle, an image can be animated with a minimal disturbance to the background.

Active Video Image Selection

In the case of active video images, depending on the type of graphics capture card and processor, the selection process may involve capturing a static image first and then manipulating the static bitmap in much the same fashion demonstrated in the *Target* demo discussed in this chapter.

In other cases, where multiple video planes are supported, the selection process may be accomplished by drawing the area, or other targeting information, in a separate video plane and letting the system combine the targeting information with the active video for presentation.

In the case of multiple video planes, drawing the overlay using the XOR mode still remains the fastest method of repeatedly drawing and removing targeting, selection, or region outline information.

TIP

No matter how fast the system and the video card, restoring the entire image is still time-consuming and almost always unacceptable, especially when you're working with an active video image.

Area Selection Conventions

In many drawing applications, the current convention for area selection is to draw an overlay consisting of a dotted outline with rectangular handles at the corners and centers of the sides. By placing the mouse cursor anywhere within the outline and pressing the mouse button, the selected region can be dragged to another position. On the other hand, by clicking on one of the handles, the outline can be dragged to a new size.

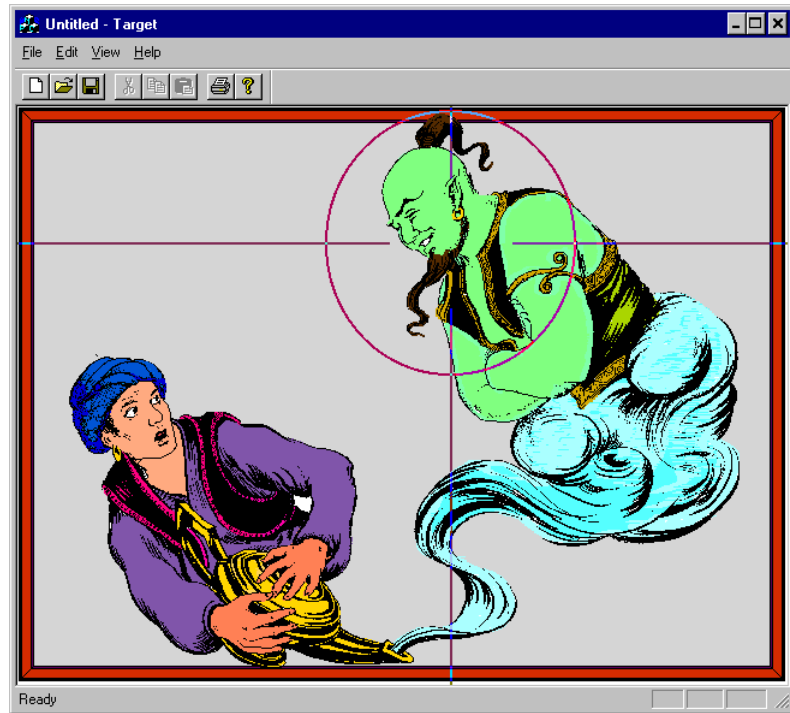
In the case of irregular areas, depending on the application, small “handles” may appear at nodes representing the vertices of a polygon outline. These handles are treated in the same fashion as a rectangular outline, permitting a vertex to be relocated. In other cases, such as in the Windows Paint program, no methods are provided for adjusting a free-form outline.

In the *Target* demo, a different set of conventions is used. For selection, you use a set of crosshairs that extend to the window margins and a circle that approximates the target area or region of interest (ROI). Figure S18.1 shows an example of a screen in the *Target* demo, with a bitmap displayed behind the target selection overlay.

This format is common in machine-vision applications, where the user is selecting an area for examination. Because the crosshairs extend to the margins of the window, they can be used to indicate a position on scales along the sides. The center of the crosshairs is left open, so that the specific target is not obscured. The circular target area marker is used to select an area for closer examination or for action by other associated tools. As an alternative, an elliptical, rather than circular, shape could be used for the target area marker.

FIGURE S18.1:

Targeting an area in a
bitmap

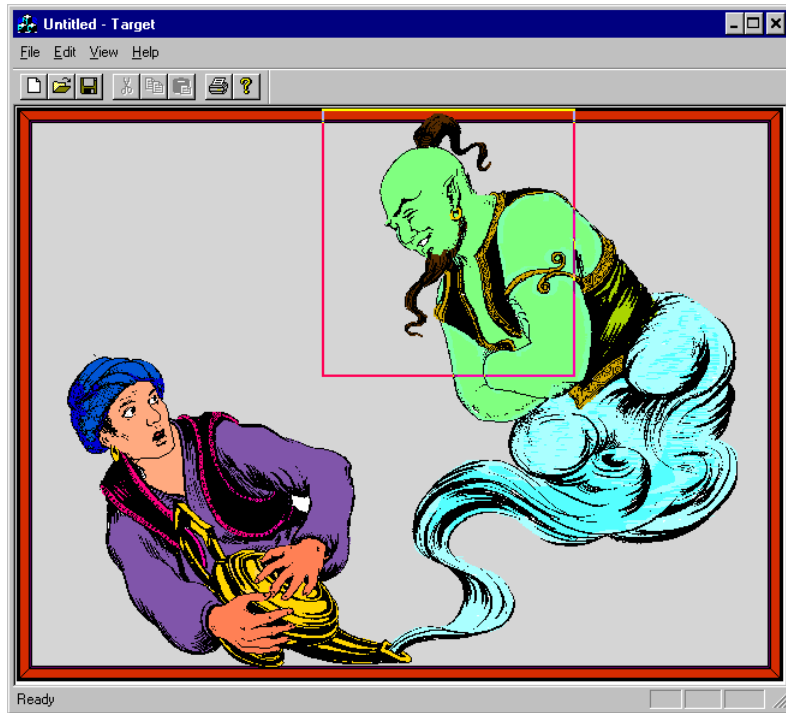


In most applications, after a user selects a rectangular or irregular region, the selection is actually moved, copied, or otherwise processed. During this processing, the common convention is to show an outline surrounding the selection. While it would be possible to capture or process a circular or irregular area, the usual practice is to process a rectangular image. This simplification is commonly used to show the actual area being processed, as well as to facilitate drag operations.

In the *Target* demo, when the right mouse button is pressed to initiate a capture (though no actual capture is done in this example), the crosshairs and circular target are replaced by a rectangle bounding but outside the region of interest. Figure S18.2 shows an example of the selection rectangle.

FIGURE S18.2:

Indicating the selected region of interest



Whether you use these conventions, or any of several others, depends on the needs of your application. There is no single set of conventions that apply to all cases and cover all requirements.

The *Target* Demo: Selecting Parts of an Image

The *Target* demo demonstrates using adjustable target overlays, testing for mouse hits with overlapping targets, and setting custom mouse cursors.

NOTE

The *Target* demo is included on the CD that accompanies this book, in the Supplement 18 "Graphics Selection Operations" folder.

Opening a Bitmap File

The *Target* demo provides an option to open a bitmap file for a background image. It uses essentially the same bitmap file and display operations demonstrated in Supplement 15 (on the CD accompanying this book). However, there are a few differences, because the version presented here relies heavily on MFC-defined classes rather than the standard APIs and conventional programming methods.

Also worthy of your attention is the single TRY...CATCH exception handler used when opening a bitmap file:

```
TRY
{
    CFile cFile( csFName, CFile::modeRead | CFile::typeBinary );
    SetCursor( LoadCursor( NULL, IDC_WAIT ) );
    ...
    read the bitmap file here
    ...
}
CATCH( CFileException, e )
{
    #ifdef _DEBUG
        afxDump << "File access failed: " << e->m_cause << "\n";
    #endif
    return FALSE;
}
END_CATCH
```

The CFile constructor (used to open the file for reading), like any class constructor, does not return an error regardless of what might go wrong. Therefore, to catch an error when opening a file in this fashion, the TRY...CATCH exception handling is required.

NOTE

C/C++ supports several forms of `try...catch` and `TRY...CATCH` exception handling, with the latter form—demonstrated in the *Target* demo—using macros. Refer to Chapter 9, “Exception Handling” for more information on using exception handling.

Responding to the Mouse

In the *Target* demo, the selection overlay must be able to respond to the mouse in several different fashions, depending on where the mouse is clicked:

- If the mouse is clicked in the center of the target, the target can be dragged to a new position without changing its size.
- If the mouse is clicked on the left or right side of the target, or on the top or the bottom of the target, the target can be resized horizontally or vertically (but not both) without changing its center position.
- If the mouse is clicked on a corner of the target—upper left, upper right, lower left, or lower right—the target can be resized both horizontally and vertically, again without changing its center position.

In each case, the circular target is the focus of these operations; the crosshairs simply follow the center position of the target. Also, the background image remains unaffected by any of these operations.

Changing the Cursor

The *Target* demo also includes provisions to change the cursor to reflect the type of operation about to occur when the left mouse button is pressed. Although the Windows GDI offers a variety of standard cursors, these are not always readily visible against a complex background (see Supplement 6 on the CD for more information about Windows cursors). The original program from which *Target* was derived provides a set of custom cursors, which have been incorporated into the demo as well.

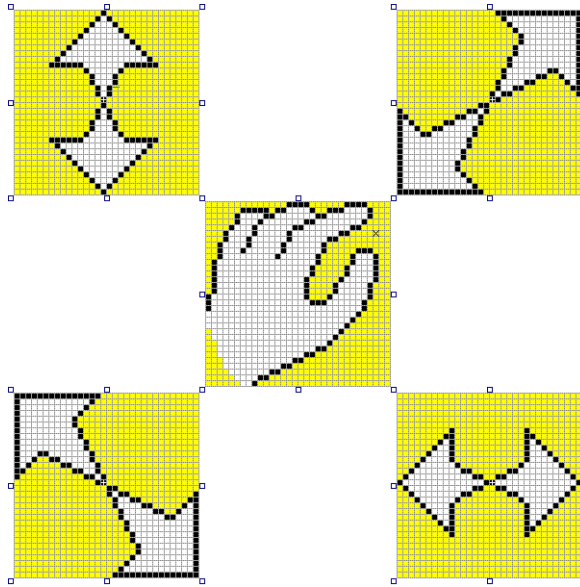
These five custom cursors appear in Figure S18.3:

- North-south cursor (`NS_CURSOR`)
- Northeast-southwest cursor (`NESW_CURSOR`)
- Hand cursor (`HAND_CURSOR`)

- Northwest-southeast cursor (NWSE_CURSOR)
- East-west cursor (EW_CURSOR)

FIGURE S18.3:

Five custom cursors



In each case, the cursor image consists of a white body with a reversed outline. A small crosshairs mark has been added to each image to identify the position of the cursor's hotspot.

Determining the Hit Position

To manage dragging and resizing operations for the target overlay, the first step is to determine where the mouse hit occurs—that is, the mouse's position when the primary mouse button was pressed.

In the `CTargetView` class, the `OnLButtonDown` method is called whenever the left (or primary) mouse button is pressed and receives two parameters, the `nFlags` parameter and the `point` argument. In this case (as in most), we ignore the `nFlags` argument, which contains status information, and rely on the `point` information, which tells us where the mouse was when the event occurred relative to the application client window.

NOTE

By default, the mouse handler causes an `OnLButtonDown` call when the left mouse button is pressed. If, however, the Mouse utility in the Control Panel has been used to swap the mouse buttons, the `OnLButtonDown` call responds to the right mouse button being pressed. For programming purposes, the primary mouse button always identifies itself with a `WM_LBUTTONDOWNxxxxx` message, and the secondary button always reports as `WM_RBUTTONDOWNxxxxx`, regardless of which physical mouse button is pressed.

Before any tests are made, the `m_nTrack` member is initialized as `NO_TARGET`. Subsequently, if a hit is identified in any target region, `m_nTrack` is reassigned a value to identify the correct region.

Also, before any other operations, the `SetCapture` method is called to ensure that mouse messages from outside the current window will still be received. The mouse capture will be released when the mouse button is released.

```
void CTargetView::OnLButtonDown(UINT nFlags, CPoint point)
{
    m_nTrack = NO_TARGET;
    SetCapture();
}
```

Next, the `m_xRadius` and `m_yRadius` members contain the size of the target ellipse and the `m_cPoint` member contains the center point. To limit the drag operation to the center of the target area, the first target rectangle is defined using two-thirds of the vertical and horizontal radii. After creating the rectangle, the `NormalizeRect` function is called, purely as a precaution, to ensure that the bottom coordinate of the rectangle is greater than the top and the right side greater than the left.

```
CPoint  cPoint( m_cPoint );
CRect   cRect( cPoint.x - ( ( m_xRadius / 3 ) * 2 ),
               cPoint.y - ( ( m_yRadius / 3 ) * 2 ),
               cPoint.x + ( ( m_xRadius / 3 ) * 2 ),
               cPoint.y + ( ( m_yRadius / 3 ) * 2 ) );

cRect.NormalizeRect();
if( cRect.PtInRect( point ) )
```

The `PtInRect` method simply returns `TRUE` if the `point` argument lies within the rectangle, or `FALSE` if not. While such a test is not difficult to perform, the provided member function is more convenient than writing a separate operation for each check that is made here.

If `PtInRect` returns `TRUE`, the `m_nTrack` member is set to `ALL`, meaning that the entire target overlay will be dragged when the next mouse moment message is received, and the hand cursor is loaded as the active cursor.

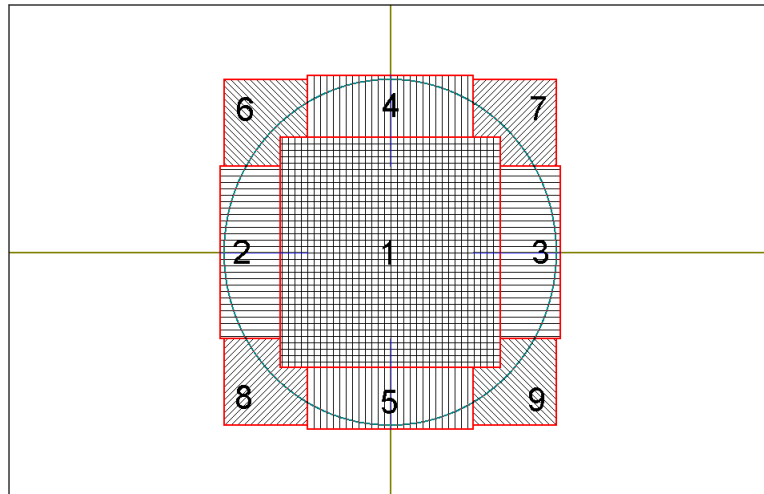
```
{  
    m_nTrack = ALL;  
    m_hCursor = SetCursor( LoadCursor( theApp.m_hInstance,  
                                     "HAND_CURSOR" ) );  
}
```

If the `PtInRect` function returns `FALSE`, the `OnLButtonDown` method continues through a series of `else` statements, testing each target area in turn. If a hit is found, it sets the `m_nTrack` variable to the appropriate operation and loads the correct cursor.

The real key here is to ensure that the target rectangles are tested in the correct order. Figure S18.4 shows nine overlapping target rectangles, numbered in the order tested.

FIGURE S18.4:

The mouse-hit target rectangles



What Figure S18.4 does not show is that region 1 overlaps regions 2, 3, 4, and 5. However, if a hit is found in the first region, no other regions are tested, making the overlapped areas irrelevant.

In like fashion, areas 1, 2, and 4 overlap area 6. But because this area is tested last, a hit will be identified for this area only if it occurs within the irregular region shown. The same holds true for the regions identified as 7, 8, and 9; each is overlapped by three other regions that are tested first.

The point is that it's unnecessary to define complex hit areas when the same task can be accomplished by testing simpler regions in the proper order. On the other hand, when it is absolutely necessary to test complex regions, you can use other methods, such as those demonstrated in Supplement 20.

Last, the `CView::OnLButtonDown` method is called to provide default handling for the mouse messages.

```

    }
    m_bDrawOverlay = TRUE;
    CView::OnLButtonDown(nFlags, point);
}

```

Because we've already provided complete handling, calling the default method is optional, but still good practice.

Once we've decided where the mouse hit occurred, the next step is to wait for the `OnMouseMove` function to be called, indicating that the mouse has moved. The `OnMouseMove` method, like the `OnLButtonDown` method, is called with `nFlags` and `point` arguments, and again, the `nFlags` argument can simply be ignored as irrelevant.

Before doing anything based on the `m_nTrack` action flag, the next step is to decide if the mouse is still in the client window. If it is not—if the mouse has been moved outside the application window—then we will release the mouse capture and do nothing.

```

void CTargetView::OnMouseMove( UINT nFlags, CPoint point )
{
    CRect      cRect;

    if( m_bDrawOverlay )
    {
        GetClientRect( cRect );
        if( ! cRect.PtInRect( point ) )
            // cursor outside of client area
        {
            m_nTrack = NO_TARGET;
            SetCursor( m_hCursor );
        }
    }
}

```

```

        ReleaseCapture();
        return;
    }

```

Next, assuming the mouse is still in the window, the response is to call the `DrawOverlay` method to erase the existing target overlay:

```

        DrawOverlay();                // erase the overlay target
        switch( m_nTrack )
        {
            case NO_TARGET:            // no action
                break;

```

If the `m_nTrack` member indicates `NO_TARGET`, then we'll take no action. If `m_nTrack` is set to `ALL`, then the `m_cPoint` member needs to be updated:

```

        case ALL:
            m_cPoint.x = max( m_xRadius, min( point.x,
                                                ( cRect.right - m_xRadius - 2 ) ) );
            m_cPoint.y = max( m_yRadius, min( point.y,
                                                ( cRect.bottom - m_yRadius - 2 ) ) );
            break;

```

Alternatively, if `m_nTrack` is set to `TOP`, the vertical radius should be adjusted according to the mouse movement:

```

        case TOP:
            m_yRadius = max( 30, m_cPoint.y - point.y );
            break;

```

The remaining case statements allow adjustments according to the quadrant selected, and the `switch` statement is followed by a test to ensure that the target is not dragged outside the client window. The bulk of these provisions, however, are routine.

The one important provision remaining is to call `DrawOverlay` a second time to redraw the target overlay at the changed position or with the changed size.

```

        ...
        DrawOverlay();                // redraw the overlay target
    }
    CView::OnMouseMove(nFlags, point);
}

```

Finally, when the mouse button is released, the `OnLButtonUp` method is called. Here, the same arguments are supplied, but now both `nFlags` and `point` can be

ignored. We don't really care where the mouse was released or what the flags were; our only interest is that the mouse button has been released. And the response to the mouse button release is simple: Reset the member flags, restore the default cursor, and release the mouse message capture.

```
void CTargetView::OnLButtonUp( UINT nFlags, CPoint point )
{
    m_bDrawOverlay = FALSE;
    m_nTrack = NO_TARGET;
    SetCursor( m_hCursor );
    ReleaseCapture();
    CView::OnLButtonUp(nFlags, point);
}
```

For the right mouse button, the provisions are even simpler: When the right mouse button is pressed, check to see if the event occurred in the target rectangle.

```
void CTargetView::OnRButtonDown(UINT nFlags, CPoint point)
{
    /*
        //=== routine to show target areas ===//
        DrawOverlay();
        DrawTargets();
        DrawOverlay();
    */

    CRect cRect( m_cPoint.x - ( ( m_xRadius / 3 ) * 2 ),
                m_cPoint.y - ( ( m_yRadius / 3 ) * 2 ),
                m_cPoint.x + ( ( m_xRadius / 3 ) * 2 ),
                m_cPoint.y + ( ( m_yRadius / 3 ) * 2 ) );

    if( cRect.PtInRect( point ) )    // is cursor in client area?
    {
```

If the mouse event is in the target, set the capture flag, call `DrawOverlay` to remove the target overlay, and then call `DrawROITarget` to create the ROI outline.

```
        m_bCapture = TRUE;
        DrawOverlay();
        SetCapture();
        DrawROITarget();
    }
    CView::OnRButtonDown(nFlags, point);
}
```


If you were tracking mouse movement while the right mouse button is pressed, this would occur in the same `OnMouseMove` method used to track movement with the primary button pressed. The only difference would be that you would need to include some provisions, such as the `m_bDrawOverlay` and `m_bCapture` flags, to determine which type of event was being tracked. Or, more directly, the `nFlags` argument accompanying the mouse-movement message could be queried to find out which mouse button was pressed or whether both buttons were pressed.

In this case, we really don't care about movement. All we're waiting for is for the right mouse button to be released.

```
void CTargetView::OnRButtonUp(UINT nFlags, CPoint point)
{
    if( m_bCapture )
    {
        m_bCapture = ! m_bCapture;
        ReleaseCapture();
        Invalidate();
        DrawOverlay();
    }
    CView::OnRButtonUp(nFlags, point);
}
```

Once the right mouse button is released, we reset our flag, call the `Invalidate` function to redraw everything in the window, and then call `DrawOverlay` to restore the target overlay. The `DrawROITarget` function does not use `R2_XORPEN`; therefore, there is no easier way to remove the ROI target rectangle.

A Note about Custom Cursors

Some of you may have noticed that the use of the `SetCapture` method in the *Target* demo appears rather redundant since, as soon as the mouse moves outside the client window, `ReleaseCapture` has been called. Why call `SetCapture` and then release it as soon as it becomes useful?

The reason is that by calling `SetCapture`, you ensure that the cursor you assign to the mouse remains the active cursor and is not replaced by the default cursor as soon as the mouse moves.

Continued on next page

The explanation for this behavior is found in the notes for the `SetCursor` function (from the MFC online documentation):

“If your application must set the cursor while it is in a window, make sure the class cursor for the specified window’s class is set to `NULL`. If the class cursor is not `NULL`, the system restores the class cursor each time the mouse is moved.”

The trick is how to set the class cursor to `NULL`. `SetCapture` provides a convenient alternative in this instance. However, the proper way to set custom cursors for a window under MFC is to intercept the `PreCreateWindow` function and to modify the `WNDCLASS` member of the `CREATESTRUCT` argument, setting the `hCursor` member to `NULL`. The revised `WNDCLASS` structure, however, must be registered before use through the `RegisterClass` function.

The long and the short of this is that setting custom cursors is not conveniently accomplished.

Other Methods of Interest

A provision has also been included in the source code to draw the several target areas used to test for mouse hits. This provision is found in the `DrawTargets` methods and was used to create the illustration in Figure S18.4. This provision can be enabled in the `OnRButtonDown` method.

Also of interest are the `OnFileOpen`, `ReadBitmap`, and `OnDraw` methods used in the *Target* demo. These parallel earlier examples but offer new versions using MFC classes and methods in place of some of the API functions and conventional operations illustrated in previous chapters.

The complete target drawing and mouse-hit recognition operations are found in the `TargetView.h` and `.CPP` files, which are part of the *Target* demo included on the CD accompanying this book.

Summary

In Supplement 18, we've taken a quick look at selecting an area within an image and creating a non-destructive overlay. We've also looked at custom-cursor and mouse-hit testing, as well as accessing and displaying a bitmap.

Next, in Supplement 19, "Interactive Images," we'll examine methods of creating interactive images, principally as maps but using techniques that could be adapted to any of a variety of images.

S U P P L E M E N T

N I N E T E E N

S19

Interactive Images

- Color keying events
- Drunkard's walk algorithm
- Memory map keys

In the previous supplement, we examined some methods for selecting sections, or regions of interest, within a bitmap image. The methods discussed there work well with simple bitmaps, but they are not suitable for bitmaps that contain more complex shapes. In this supplement, we'll look at several methods of mapping mouse events to complex bitmap regions.

Complex Regions in Interactive Images

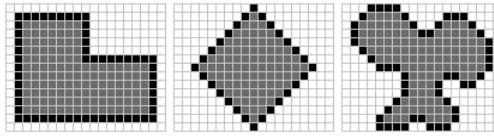
A major element, and a major problem, in many graphics applications is identifying the location within an image where an event, such as a mouse click, has occurred. If you're interested in only the window coordinates where the event occurred, this information is supplied in the `lParam` argument accompanying a `WM_XBUTTONDOWN` message or, using MFC, in the `point` argument passed to the `OnButtonDown` and `OnMouseMove` methods. However, determining where a mouse click has occurred in relation to a displayed bitmap or some other region defined on the screen is a more difficult matter, particularly when the region is not conveniently defined by a series of bounding coordinates.

For simple rectangular shapes, you can use the `PtInRect` method, as explained in Supplement 18. However, for any other shapes this method fails, and a new approach is required. For example, consider Figure S19.1, where three shapes are depicted representing possible screen areas:

- The first shape (region) at the left could be defined using a half-dozen coordinate pairs, one pair for each vertex. You could then identify a mouse event occurring within the bounded region by testing the area as two rectangular regions.
- The center region in Figure S19.1 is a simpler shape, with only four vertices. But it is also more complex, because the boundary lines are diagonal rather than rectilinear. In this case, a more complex test is required to calculate where the edges lie in relation to the mouse-click event.
- The third region, at the right, is the most complex of all. Following conventional processes, it requires a relatively large number of coordinates describing the convolutions followed by the area's outline.

FIGURE S19.1:

Three bounded regions



Methods for Identifying Regions and Enclosures

Although it's possible to create custom algorithms tailored for specific individual types of images, for a generic mapping process a different approach to identifying regions and enclosures is desirable. You can use a number of processes for this purpose, as explained in the following sections.

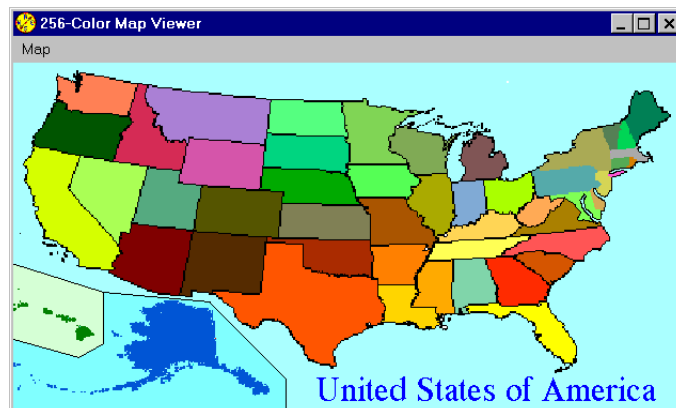
Identifying Regions by Color

One identification approach involves color keying. This approach is demonstrated in the *MapDemo* program discussed in this supplement. The demo uses the USMAP01 bitmap.

Identification by color matching relies on the fact that each region (each state) in the United States map depicted by the USMAP01 bitmap possesses a unique color value. Figure S19.2 shows the USMAP01 bitmap image displaying the contiguous United States with Alaska and Hawaii inserted at the lower left.

FIGURE S19.2:

The USMAP01 bitmap



The *MapDemo* program contains a lookup table that matches each state's color with the state's name; the color choices themselves are quite arbitrary.

This approach has a few obvious restrictions, including being limited to use on systems with capacities to display more than 16 colors and requiring that the image consist of areas of continuous color. Also, any two areas with the same color value will be identified as the same region. Overall, however, it is a practical and useful method of identifying a large number of irregular regions.

Using a Hidden Color Map

Another identification process involves using a memory-map color mask to determine where events have occurred in a bitmap that is not composed of contiguous color regions. This method uses a second color map that is not displayed; that is, it is created in a memory context rather than a display context.

This second map contains the color-identification values for the primary bitmap that appears on the screen. Then, when the mouse is used to select a point on the displayed bitmap, you look for the pixel value in the hidden bitmap and match this color to the keys.

Using the Drunkard's Walk Algorithm

An algorithm that is particularly useful for identifying irregular regions is the drunkard's walk algorithm, titled thus because the search pattern follows a trace reminiscent of an inebriated and staggering pedestrian. (This type of motion is also known as Brownian motion, as exhibited by microscopic particles subject to thermal agitation.)

Where a drunkard—or a microscopic particle—simply continues indefinitely, the drunkard's walk algorithm executes a test after each staggering step (rather like a drunkard searching for any lamppost in reach) to determine whether it has reached an identifiable point, and halts when such an encounter occurs. These target points are assigned locations within each enclosed region and are indexed to uniquely identify the region.

The drunkard's walk algorithm begins at a point indicated by the mouse click and proceeds in any arbitrary direction until one of these events occurs:

- A boundary is reached.

- A random instruction instigates a change in direction.
- The drunkard's search reaches a target coordinate identifying the enclosed region.

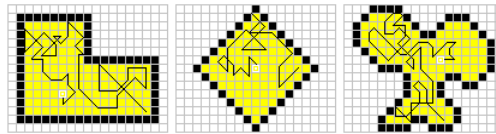
In the first case, reaching a boundary identified either by a change in color (or in the *MapDemo* program, by a specific, preselected border color), the drunkard's path simply bounces or reverses. The drunkard then retraces its path until the second instance (a random change) forces a new direction or until, ultimately, the path intersects an identifiable point.

In the second case, a pseudo-random generator initiates a change in direction every ten steps, on average. Although in one chance out of eight this chance is not a change at all, the overall effect is to trace paths with an average length of 10 steps (or, in this case pixels) between changes in direction.

The third case is illustrated in Figure S19.3. This figure shows the same three regions illustrated in Figure S19.1, but this time with a drunkard's walk trace, which ends by intersecting the desired target coordinates (shown as a small box outline).

FIGURE S19.3:

The drunkard's walk search



Using a Recursive Search

An alternative to the drunkard's walk algorithm is a recursive search algorithm. This algorithm begins, from the initial coordinates, by initiating a recursive search. For example, the recursive search might begin by searching to the immediate left, then down, then up, and finally to the right, with each point searched initiating a further recursive search in the same direction until a border is reached or the target is found.

For example, assume a recursive search beginning at point 100,100. The recursive process calls itself, first with the coordinates for the point to the immediate left (99,100). When this first search returns, assuming that the target point has not been found, the next search will be up (100,99), then down (100,101), and then right (101,100).

However, the first search at (99,100) initiates its own searches at (98,100), (99,99), (99,101), and (100,100), which is the same point where the search started. And each of these searches executes its own recursive search.

But since each search is looking for either a border, which terminates further recursion in that branch, or the target point, which also terminates recursion, this algorithm is not infinitely recursive. But, even with finite recursion, the number of active recursions does increase geometrically. Furthermore, each recursion requires its own register values to be pushed onto the stack, and this can very quickly lead to stack overflow. Even if the recursive procedure is very carefully designed, this can still be a problem.

An advantage of this approach is that a well-designed recursive search is absolutely certain to succeed. It will find the target location, even if it must check absolutely every location within a region. But a well-designed recursive search is not necessarily any faster than a drunkard's walk search, and it does consume considerably more of the system's resources. At its worst, the drunkard's walk algorithm is sometimes a bit slower but only rarely and randomly so. In general, the drunkard's walk algorithm tends to be faster, as well as more efficient in overall usage of system resources and, most important, of CPU time.

Finally, on the basis of simple aesthetics, the drunkard's walk algorithm is far more satisfying to the soul (the programmer's, at least, if not the machine's) than the stolidly pedestrian recursive search. After all, getting there is half the fun, isn't it?

The MapDemo Program: Identifying Event Locations in a Bitmap

As explained earlier in the supplement, the *MapDemo* program uses the USMAP01 bitmap, which is a map of the United States. This program demonstrates the use of the color matching and drunkard's walk algorithm techniques for event identification.

NOTE

The upper New England states (which appear relatively small in the USMAP01 bitmap) are displayed separately in the USMAP02 bitmap and are used to demonstrate the second area-identification algorithm. The USMAP02 image can be selected either by clicking on the upper New England states in USMAP01 or through the Map menu.

Color Matching

Within the *MapDemo* program, while the USMAP01 bitmap is displayed, a `WM_LBUTTONDOWN` message calls the `ColorCheckMap` function, passing three parameters: the window handle (`hwnd`) and the two mouse-click coordinates derived from the `lParam` argument accompanying the mouse-button event message.

```
case WM_LBUTTONDOWN:
    ...
    CoordCheckMap( hwnd, LOWORD( lParam ),
                  HIWORD( lParam ) );
    break;
```

The mouse-click x-axis coordinate is reported in the low word of `lParam`, and the high word reports the y-axis coordinate.

The `ColorCheckMap` function appears to be a simple process, but it requires a bit of finesse to comply with Windows' requirements.

```
void ColorCheckMap( HWND hwnd, WORD xCoord, WORD yCoord )
{
    HDC      hdc;
    DWORD    RColor;
    WORD     SColor;
    int      i;

    ...
    hdc = GetDC( hwnd );
    RColor = GetPixel( hdc, xCoord, yCoord );
    // need RGB palette-relative color value
    ReleaseDC( hwnd, hdc );
```

The `GetPixel` function returns a `DWORD` value containing the RGB color value for the selected pixel in the form `0x00rrggbb`. However, while this is the color of the pixel itself, the data identifying the several states consists of the simpler palette index values rather than their RGB equivalents. Ergo, the next task is to match the color returned to the bitmap's palette, retrieving the palette index.

To accomplish this, the first requirement is to lock the pointer to the global palette information (`pGLP`) and then lock a handle to the logical palette (`hGPal`) before calling the `CreatePalette` and `RealizePalette` functions to temporarily recreate and activate the bitmap palette.

```
LocalLock( hGLP );
LocalLock( hGPal );
```

```

        // lock and create palette for reference
hGPal = CreatePalette( pGLP );
if( hGPal == NULL ) ErrorMsg( "Palette not found!" );
        // make palette active for device context
RealizePalette( hdc );
        // get palette index for comparison
SColor = GetNearestPaletteIndex( hGPal, (COLORREF) RColor );
        // unlock everything but don't delete palette
LocalUnlock( hGLP );
LocalUnlock( hGPal );

```

Finally, after the palette is created (or recreated), the `GetNearestPaletteIndex` function returns the palette index value as `SColor`. Of course, before finishing, the two memory locks on the global and recreated palettes should be released. Neither, however, should be freed from memory since they may be needed again.

Once the palette index has been retrieved, a pair of simple loops is all that is required to identify the corresponding state or, in the case of the upper New England states, to display the `USMAP02` bitmap14.

```

for( i=0; i<12; i++ )
    if( SColor == NewEngland[i] )
        // if this is any New England state, switch maps
        PostMessage( hwnd, WM_COMMAND, IDM_MAP2, 0L );
for( i=0; i<=StateColors; i++ )
    if( SColor == CState[i].Color )
        LocationMsg( CState[i].State );
MessageBeep( MB_ICONASTERISK );
}

```

Once the state or area is identified, a variety of other responses can be implemented as elaborately or as simply as desired. In this demo, a simple pop-up dialog box with a “Welcome to the great state of xxxxxx” message appears, identifying the state selected.

The data matching the states and colors is provided by a simple structure listing these by name and palette index. An abbreviated sample follows:

```

ColorState CState[StateColors] =
{
    "Arizona",      2,      "New Mexico",      7,
    "Oklahoma",     9,      "Georgia",         11,
    "Oregon",       12,     "Colorado",        13,
    "Missouri",     15,     "South Carolina",  16,
    "Texas",        17,     "Hawaii",          18,
    ...

```

Implementing the Drunkard's Walk Search

Implementing the drunkard's walk search algorithm (CoordCheckMap) is a relatively simple task. As in the ColorCheckMap function, this search is called with three arguments: the window handle and the x- and y-axis coordinates reported by the mouse-click event message.

```
void CoordCheckMap( HWND hwnd, WORD xCoord, WORD yCoord )
{
    BOOL    Done = FALSE, Reverse;
    HDC     hdc;
    WORD    i;
    int     j, k, x, y;

    ...
    randomize();
    hdc = GetDC( hwnd );
    x = random(3)-1;
    y = random(3)-1;
```

Initially, CoordMapCheck retrieves the device-context handle for the window displaying the map and selects step directions (x, y) in the range -1 to 1. Since these are used to increment the xCoord and yCoord values, the result is a search track beginning in one of eight compass directions (N, NE, E, SE, S, SW, W, NW).

NOTE

There is also one chance in nine that the initial search step will be (0,0), which is no search at all. This will, however, correct itself automatically when the next random search direction is selected.

Before the search is initiated, the first step is to check the present coordinates against a loop testing all of the identified coordinate pairs. Also, rather than requiring a perfect hit on the target coordinates, the actual test accepts any point that is within ten pixels of the target (total offset on both axes). As in a game of horseshoes, close *does* count and an exact match is not required, simplifying the search.

```
do
{
    for( i=0; i<AllCoords; i++ )
    {
        if( ( abs( xCoord - Coord[i].xPos ) +
```

```

        abs( yCoord - Coord[i].yPos ) ) < 10 )
    {
        if( i < StateCoords )
            LocationMsg( Coord[i].State );
        else
            PostMessage( hwnd, WM_COMMAND, IDM_MAP1, 0L );
        Done = TRUE;
    } }

```

If the current coordinates are within a total of 10 units of the target coordinates, a match is simply assumed. The hidden assumption here is that no target point is located within 10 pixels of a border; otherwise, it could be misidentified by being detected from the wrong side of the border.

In other cases, a closer (or looser) match might be appropriate, so you would adjust the algorithm accordingly. For the present, a range of 10 pixels is adequate for the purpose.

State coordinate pairs in the *MapDemo* program are identified, together with the appropriate state names, as a simple structure table:

```

CoordState Coord[AllCoords] =
{ "Connecticut", 267, 208, "Delaware", 202, 311,
  ...,
  "Vermont", 238, 132, "RETURN", 14, 337,
  "RETURN", 331, 290, "RETURN", 122, 81 };

```

In addition to the state coordinates, the program provides three area coordinates that do not fall within a specific state: one below the upper New England states, and two in the blank areas surrounding these states. Intersecting any of these three locations sends the application back to the USMAP01 display.

Alternatively, as long as a match is not found, the next test is to determine whether the bitmap borders have been reached:

```

Reverse = FALSE;
if( ( xCoord >= bmWidth-1 ) || ( xCoord <= 1 ) ||
    ( yCoord >= bmHeight-1 ) || ( yCoord <= 1 ) )
    Reverse = TRUE;

```

If, for any reason, the search is approaching the bitmap border, the Boolean *Reverse* flag is set and, subsequently, will reverse the search direction. Without this provision, the usual result under Windows will be a system application error, often trashing the system memory as well.

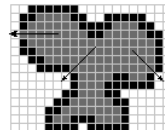
Next, as long as the search remains away from the bitmap border, a second loop executes a check of the immediate vicinity, searching for pixels identifying a border encounter and, again, reversing direction if a border is found.

```
else
    for( j=-1; j<2; j++ )
        for( k=-1; k<2; k++ )
            if( GetPixel( hdc, xCoord+j, yCoord+k ) == BORDER )
                Reverse = TRUE;
```

The program could have executed a simple, straight-ahead search, but this would have one fairly dangerous flaw: Narrow borders can leak through either single-pixel gaps or diagonal “pores,” both of which are illustrated in Figure S19.4.

FIGURE S19.4:

Leaky borders using the drunkard’s walk algorithm



Here, a gap in the left border is one potential leak where a search trace could escape. Two other locations show pores where a diagonal search trace could escape. The remaining potential gaps in the original have been blocked in this version. Going through a bitmap looking for potential problems of this type, however, is a tedious process and prone to error. Instead, the broad area-checking provisions in the preceding code accommodate a much less rigorous border condition.

Finally, if any of the preceding tests have set the Reverse flag, both the x and y increment variables are inverted by multiplying by -1.

```
if( Reverse )
{
    x *= -1;
    y *= -1;
}
else
if( ! random(10) )
{
    x = random(3)-1;
    y = random(3)-1;
}
```

Alternatively, if no reverse condition has been encountered, a simple random test is used to change the search direction on a 1-in-10 chance. Like the original one, the new search direction is selected randomly.

NOTE

A provision has been included in *MapDemo* to render the search trace visible by drawing a white dot at each step. This is implemented quite simply as:

```
//=== option to trace random walk algorithm =====
                SetPixel( hdc, xCoord, yCoord, 0x00FFFFFF );
//=====
```

To disable this trace provision, simply comment out this line of code.

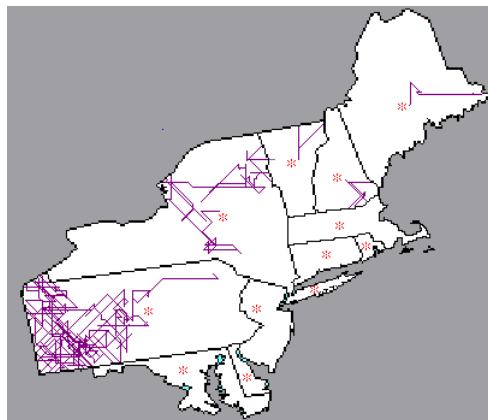
Last, the current x and y incremental values are added to the xCoord and yCoord values before the `do...while` loop continues.

```
        xCoord += x;
        yCoord += y;
    }
    while( ! Done );
    ...
    return;
}
```

The illustration in Figure S19.5 shows several drunkard's walk searches executed in various New England states. The illustration was created by having the search paint its own trail markers. After the screen image was captured, the fill colors for each state were replaced by white, and the target points (which appear as small red dots in the map image) were replaced by asterisks for easier identification.

FIGURE S19.5:

Drunkard's walk searches
in New England



NOTE

The drunkard's walk algorithm is not without an occasional shortcoming. Stuart Ozer, who was my technical reviewer for the Windows 3.1 version of an earlier volume, reported finding a starting point on Cape Cod from which the algorithm required 10 minutes or more to identify Massachusetts. Such a flaw could be blamed on the geometry of the search versus the boundary configuration, or it could be simply bad luck in the pseudo-random number sequence directing the search.

Using a Hidden Map

There may be occasions when you want to identify areas without having them visible on the displayed image. For example, in the case of the United States map, you might wish to display a topographic or meteorological map of the country without delineating the states, which would mean no borders.

An alternative approach that will allow you to still be able to identify the states would be to use a second map image—the same size as the first—that does not appear on the screen. You would create this second map in a memory context and, when the mouse is used to select a point on the displayed bitmap, the search would be executed from the corresponding point on the hidden bitmap.

Using Direct Coordinate Searches

Although it may appear odd or even inefficient, the drunkard's walk algorithm is, overall, a very fast technique for determining a regional location. There is, however, one alternative that, on first consideration, sometimes appears more efficient: Search the coordinate list for the coordinate pair closest to the starting point, then look for a border between the two points.

The reasons for this second step are simple but are best illustrated by an example. Consider the states of Pennsylvania and New Jersey and assume that the coordinate pairs for each are located at, approximately, Altoona (Pennsylvania) and East Brunswick (New Jersey), placing each location roughly at the center of the state.

A mouse click in the region of Philadelphia (Pennsylvania) would not, however, select the Altoona coordinates as nearest because the East Brunswick coordinates in New Jersey are considerably closer.

Of course, searching for a border between the initial point and the closest target would identify the problem and allow another search for the next closest coordinates (and so on) to proceed. But the search for a border is almost as time-consuming as a simple drunkard's walk and is more difficult to program reliably.

Now suppose that the located coordinate set is correct—that it lies in the same state or region as the mouse click—but that a straight line between the origin and the coordinate point must cross two borders. Sound unlikely? It isn't. For example, look at Figure S19.5, where the state of Maryland (at the bottom of the map) is virtually split in two by the Chesapeake Bay. At the same time, Long Island and the mainland portion of the state of New York are technically all one region but, physically, are two separate areas on the map. In this instance, neither of the algorithms discussed—the drunkard's walk or the closest points with border-crossing tests—is adequate.

The solution for both the cases of discontinuous or extremely convoluted areas is to provide more than one coordinate point. In the *MapDemo* program, the USMAP02 provides two coordinate points to identify New York: one on Long Island and one on the mainland.

For Maryland, a second point located across the Chesapeake Bay would simplify searching and would also prevent an error that is present in the current version: Selecting a point across Chesapeake Bay is very likely to locate Delaware, not Maryland.

NOTE

It would also be possible to provide a larger number of coordinate point targets in each area and, granted, this should ensure very fast searches. The only drawback would be the effort of building tables of points and ensuring that there were no errors in the result. However, the trade-off in speed—which is minimal from the user's viewpoint—hardly seems worth the effort.

Summary

In this supplement, we've looked at several methods for working with interactive, complex bitmaps. The *MapDemo* program, included on the CD accompanying this book, demonstrates two methods for mapping mouse events to complex

bitmap regions. You can experiment with these and the other methods mentioned here to see how they suit the needs of your applications.

Next, in Supplement 20, “Graphics Simulations,” we will look at how graphics simulations work and methods for developing your own simulation applications.

S U P P L E M E N T

T W E N T Y

S20

Graphics Simulations

- A synthetic cosmos for modeling physical interactions
- Variable timing for simulated events
- Choices for simplifying simulations
- Mechanical simulations
- Theoretical system simulations

Over the past two decades, computers have revolutionized more of our world than most people realize. Granted, the Internet is the current buzzword, and most people are aware that computers are used for special effects in movies and commercials on television. Some people are even familiar with using computers to study fluid dynamics, weather patterns, engineering structures, and other technical subjects. But these are only a few of the areas where computers have radically changed the traditional arts, crafts, and sciences. A complete list of affected areas would fill a large book, which would be out of date long before it could be published.

But this supplement is not about the computer revolution. It's about an area that did not even exist, with a few modest exceptions, prior to computers: the field of graphics simulations or, more accurately, the mathematical simulation of dynamic systems in general, including both physical and nonphysical systems.

Using Graphics in Simulations

Simulations do not necessarily require graphics, and in some cases would be slowed down by graphics. However, we have a very human desire to see what is happening while it happens, rather than reading about the results afterwards. As an example, the *Forest* demo discussed in this chapter displays a small universe of 10,000 acres, simulating the growth of trees.

The simulation begins with bare ground that is randomly seeded with 100 starts (seedlings). As the simulation progresses, the various wooded areas grow, age, and propagate, spreading trees to new areas.

Of course, if this were the extent of the simulation, this mini-universe would simply fill with trees until there was no bare ground left. The result could be derived simply by calculating the average time necessary to fill the forest. This simulation, however, is not so limited.

Instead, as a defined area of tree population ages, the trees eventually die, rot, and leave a new plot of bare ground. At the same time, in emulation of the real world, the simulated forest is subject to fires. And once a fire starts, it spreads. The older trees are easiest to ignite, and changing wind patterns affect the spread of the fire.

Overall, if the simulation's only output were a statistical report listing the forested acres for each year, we might learn almost as much. But "almost as much" is not the same as watching it happen. By watching the forest grow, burn, and reseed itself, we gain some small measure of understanding of two new and very important elements: the patterns of growth and death, and the way that changing the parameters affects not only the end results but also the patterns themselves.

NOTE

If you have any doubts about the relative importance of simple statistical results versus patterns, consider the extreme examples of any fractal algorithm. In fractal calculations, such as the Henon Attractor or the Malthusian equation (another famous simulation), statistical results reveal almost nothing; the patterns, visible only when plotted graphically, reveal everything.

Some Background on Computer Simulations

In the ages B.C. (Before Computers), the sheer volume of calculations required for simulations ruled out modeling even the simplest systems unless some measurable physical analog could be employed. In some cases, there were alternatives. Physical erosion was relatively easy to study using a slant box of sand and a water source. Minimal route-mapping problems could be solved using soap films. And many ballistic and navigational problems were attacked using electronic (and some mechanical) analog systems.

As for more general simulations, however, the Life program was played out with paper and pencil, usually by students who might have better spent their time studying. This was roughly the practical limit for an unaided human. (For your amusement, the Life program in electronic form is included on the CD accompanying this book.)

In the Life program, the "world" consists of a grid that, for convenience, is finite but unbounded (see "The Toroidal Model" later in this chapter) and each grid location may initially be "alive" or "dead." Provisions are included in the demo to "seed" the grid randomly or to create an initial configuration known as a *launcher* that will generate two child *flyers*, which will fly across the screen on a diagonal path.

Continued on next page

At intervals of $\frac{1}{2}$ second, a new generation of Life is calculated according to the current state of the “world” and four simple rules:

1. Any “alive” location that has more than three neighbors dies of overcrowding in the next generation.
2. Any “alive” location that has fewer than two neighbors dies of loneliness in the next generation.
3. Any location—“alive” or “dead”—that has exactly three neighbors will be alive in the next generation.
4. Any location—“alive” or “dead”—that has exactly two neighbors remains unchanged in the next generation.

Advancing beyond these three simple rules governing the Life program and expanding beyond what is, essentially, a very small universe, the complexity and volume of the calculations required for most simulations have simply overwhelmed both human patience and practical capacities. Some few individuals have accomplished prodigious feats of cogitation and calculation, such as the compilation of the Rudolphine Tables (Kepler) or calculation of the trigonometric functions (Napier), but these are exceptions as well as monumental endeavors. (The Aztec calendar might also qualify but was almost certainly a prolonged group effort.)

Thus, for the most part, simulations of any complexity have waited for the advent of our newest and most powerful tool: the computer. Using this tool, we are now able to study—through simulation—systems that we could only theorize about previously.

All of this says nothing about the accuracy of our simulations, but it does permit testing our theories against actual performance. Therefore, if your theory holds that playing to fill an inside straight is better than folding on the sixth card, you can create a simulation to test this theory faster and more accurately (as well as more cheaply) than testing the theory at Saturday night poker games. (Of course, this question can also be settled by probability theory without requiring simulations, but we won’t go into that here.)

Creating a Dynamic (Memory) Cosmos

The *Forest* demo demonstrates the creation of a synthetic cosmos. By intention, the forest exists only as a shadow, mimicking reality without requiring the complexity of rules (natural laws) that govern what we familiarly consider reality.

Instead, the simulation uses simpler rules that can be manipulated, compressed, and studied. Thus, by analogy and experimentation, we are able to better understand the complexities of reality.

Rather than modeling the growth and complexity of individual trees (along with the weather patterns, soil composition, and a myriad of other factors) and repeating this for the thousands of acres of trees composing the forest, the forest is calculated as areas following a simple statistical growth pattern with a uniform composition within the area. In this fashion, we *can* see the forest for the trees; we are able to look at the forest as an entity while ignoring the trees themselves.

The Forest Cosmos Size

Our simulated forest exists in a cosmos consisting of a scant 10,000 units (100 by 100) that—for convenience only—are referred to as acres. Because this is a simple simulation, the essential status for each unit is stored in an array of BYTE.

For convenience, two arrays are used, permitting the second array to be updated by reference to the first and to then replace the first array. In this fashion, the first array, which holds the prior status, is not affected by changes that would produce recursive effects.

As you know by now, although DOS and Windows 3.x impose limits on array sizes of a mere 64KB, Windows 2000 (using 32-bit addressing) has revoked this limitation in favor of a theoretical array size of 4GB. Of course, we still face physical limitations imposed by the amount of memory available; even on small systems, however, this is a considerable increase in freedom.

NOTE

If you need to use extremely large arrays, in sizes beyond available memory limits, you can use disk files as an extension of RAM. Unfortunately, this approach has the disadvantages of being relatively slow and somewhat cumbersome.

Rules of the Forest Cosmos

“Had I been present at the creation, I might have offered the creator much valuable advice.” — Remark attributed to Alphonso the YYs

The Forest cosmos is governed by a series of relatively simple rules, which appear as numerical algorithms within the program:

- The Forest cosmos begins as bare ground and is randomly seeded, initially, with 100 plantings.
- In subsequent cycles (years), the planting age is shown by changing colors.
- After a minimum of five cycles, the forest plots are developed well enough to propagate and, if adjacent plots are bare, may seed these areas, initiating new growth.
- Old-growth acres (arbitrarily those over 11 years old) are susceptible to natural death. A simple simulation provides for old age and other causes. As with natural forests, however, this is a minor element and affects approximately one percent of the forest.
- Fire is a major effect in the Forest cosmos, just as it is in real-world forests. For simplicity, only one fire can start during any cycle. Minor fires are not simulated, but a fire may spread to adjacent acreage.
- Fires die out when their fuel is exhausted, but they are also affected by wind direction and speed.
- Fires can spread only to mature acreage; young plots are not affected (under the assumption that young trees are scattered and little deadwood is available to fuel a major burn).

Given these relatively simple rules, the Forest cosmos simulates the same patterns of growth and death exhibited by real forests. And a correspondence in patterns is the hallmark by which a simulation is tested.

Handling Boundary Problems: Creating a Closed, Unbounded Cosmos

Any simulation that re-creates or models a subset of a larger reality is subject to a boundary problem in one form or another. When any area is subject to effects from surrounding areas—and, naturally, vice versa—boundary problems are found in simulations whenever there are no adjacent areas (in the simulation). One option, of course, is to have boundary areas that are relatively static, that are not affected by the simulation area, or that do not have an effect on the simulation

area. For the Forest simulation, for example, we might use a boundary consisting of a rocky desert or ocean (simulated) such that these boundary areas were irrelevant to the simulation.

Alternately, our boundary areas might consist of rules simulating a surrounding cosmos or simulating a barrier of some theoretical sort. Exactly how complex you wish to make the boundary is, of course, subject to the needs of your simulation. Or, as a further alternative, we can simply create a cosmology where the boundary does not exist.

In the Forest cosmos, the boundary problem is avoided by the simple expedient of making the cosmos a closed, unbounded universe. What appears to be the left edge of the map actually adjoins the right edge; the top edge of the map joins and continues at the bottom. Topologically, this type of closure is the equivalent of a *toroidal* surface (a doughnut or an inner tube are physical examples of a toroidal surface). Although toroidal surfaces are not commonly encountered in our universe (at least, not on any macrocosmic scale), this is a popular method of avoiding boundary problems in simulations.

There are other, more complex methods for dealing with boundary problems. For example, a method popular in the study of the physical universe involves using a spherical surface. Another even more complicated and computation-intensive approach involves using algorithms to simulate the effects of areas outside the actual simulation boundaries.

For most planar simulations, the toroidal universe provides the simplest approach and the fastest computational results. Furthermore, if your simulated cosmos is not planar but a volume, such as a fluid or gaseous volume, this same practice can be extended to create a hypertoroid in cybernetically four-dimensional space.

The Toroidal Model

The practice of creating or simulating a closed but unbounded cosmos in cyberspace is both simple and complex. On the simple side, because the data describing a simulation is stored in one or more arrays or matrixes, the primary consideration in using the toroidal surface model is to test all coordinate references (that is, references to array data) and to provide adjustments for references that fall outside the array limits, thus “wrapping” the index back into the array from “the other side.”

On the complex side, although simple rectilinear offsets are easily converted, operations involving vectors, angles, or curves are not always easily handled. When operations of this or a similar type are necessary, the simplest approach is to use a separate matrix where the operation can be carried out without crossing a boundary. The results can be mapped, using whatever offsets and adjustments are necessary, into the simulation space.

In spatial terms, the most important element is to make sure that all operations that wrap across an array boundary are correctly adjusted for the wrap. Failure to do so can have strange and interesting results that are not always easy to identify or recognize.

Using Colors in Simulations

One principal characteristic of graphic simulations is the use of color to make information clear. In many cases, commonly referred to as *false-color mapping* or *false-color imaging*, color assignments are arbitrary and have no real-world relation to the source or the data.

For example, false-color imaging is often used in astronomy to “translate” radio-frequency images for visual presentation. The translation involved can take several different forms, including using color to represent intensities, radio frequencies, densities, or even gravitational gradations—none of which have any direct correspondence to the visual spectrum.

Another mapping format uses colors that are chosen to represent approximate analogs of the data. An example of this latter approach is used in the Forest simulation. Bare ground is represented by browns, various stages of forest growth by shades of green, fires and embers by reds, and ashy ground by grays.

You can also use a combination of both representational color and false-color coding. This approach generally involves switching between display formats, showing first one information set and then another. For example, you could create a switched display for the Forest program by adding provisions to show simulated rainfall patterns, temperature profiles, or soil-composition characteristics (extensions you can experiment with yourself).

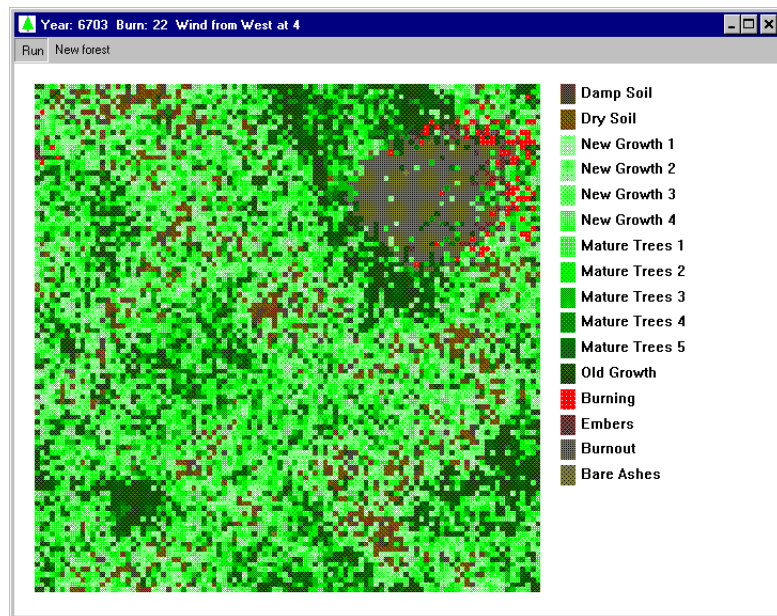
The Forest Demo: Operating the Simulation

Deciding how to set up a simulation is the first step; coding the simulation is the second. Both steps require provisions for a variety of circumstances.

As an example, Figure S20.1 shows the Forest cosmos some 6,703 years after seeding and 22 days into a major burn-off. The illustrated burn began somewhere in the northwest of the display but has not spread too widely, despite a current wind from the west at a strength of 4 (the maximum is 5). Allowing the simulation to continue, the fire revives as the winds change direction and finally burns itself out after 73 days.

FIGURE S20.1:

Here the forest is 6,703 years old, 22 days into a burn condition, with the wind from the west at a strength of 4.



All of these events, of course, exist only in the computer's memory and result from pseudo-random number sequences. Nonetheless, they provide a faithful emulation of patterns of growth and burn-off that have been observed in natural forests.

Defining the Color Palette

In the *Forest* demo, a palette of 16 colors has been defined. Two of these represent bare ground (light brown and dark brown). Ten shades of green represent trees at various ages. Light and dark red represent fires and embers, respectively. A dark gray is used to represent freshly burnt, ashy ground. A third brown is used for ground left bare by a fire.

The palette colors are defined in `Forest.h` as RGB values, each with a corresponding integer constant as a convenient identifier. (These latter elements are for the programmer's convenience only—but, then, what is more important?)

However, although these RGB values are defined for each color, this palette is never activated for the device context. Instead, while the RGB colors are referenced as drawing colors, Windows is allowed to dither the existing default palette colors in drawing the simulation map.

NOTE

The decision to permit dithering instead of activating the color palette is arbitrary, but this method does show, even on SVGA systems, how a similar display might appear on standard VGA systems. The alternative of activating the defined palette colors is left as an exercise for the reader (and is demonstrated numerous times in other sample programs accompanying this book).

Initiating the Cosmos

Lacking the resources to initiate a primal fireball and to then wait for nature to evolve a life-form from the first primordial globule, the first provision in the *Forest* simulation is to set the initial conditions for the simulation. This is accomplished in two steps, beginning in response to the `WM_CREATE` message.

```
case WM_CREATE:
    srand( (unsigned)time( NULL ) );
    ActiveTimer = 0;
    wsprintf( szBuff, szCaption, nYears );
    SetWindowText( hwnd, szBuff );
    for( i=0; i<GRID; i++ )           // init world as
        for( j=0; j<GRID; j++ )       // bare ground
            Acres[i][j] = random(2);
    break;
```

The `randomize` function ensures that new initial conditions (that is, a new pseudo-random number sequence) are used each time the program is executed.

NOTE

Depending on the compiler you are using, the pseudo-random number algorithm can be initialized in different fashions. Some compilers provide a `randomize` function which automatically calls the system clock to seed the generator. Using Visual C++, calling the `srand` function with the system time serves the same purpose.

The second provision, the double loops, sets the world to a random mixture of damp and dry soils shown by the two browns. Within the program, both are treated simply as fertile ground, without regard to their moisture content. In a more elaborate version, the two browns might be used as different conditions, representing wet and dry ground or high and low terrain. For the current version, two shades of brown are simply not as dull as a single uniform color.

The second stage of initialization occurs only when the New Forest option is selected from the menu. When this option is selected, the program resets the `nYears` variable to zero and seeds 100 random locations, setting the corresponding array elements as `iNewGrowth`.

```
case IDM_RESEED:
    srand((unsigned)time(NULL));
    nYears = 0;
    for( i=0; i<100; i++ )
    {
        xPos = random(GRID);
        yPos = random(GRID);
        Acres[xPos][yPos] = iNewGrowth;
    }
    EnableMenuItem( GetMenu(hwnd), IDM_STOPTIMER,
                    MF_ENABLED );
    PostMessage( hwnd, WM_COMMAND, IDM_STARTGROWTH,
                  0L );
    InvalidateRect( hwnd, NULL, FALSE );
    break;
```

The pull-down Run menu has two options, Start and Stop, both of which are initially disabled. After the forest is seeded, the Stop option is enabled, and a message is posted to start the `ID_GROWTH` timer. Last, the `InvalidateRect` function is called to repaint the window.

Setting Variable Time for Events

After the initial conditions for a simulation are set (your cosmos is created), normally the next step is to initiate a sequence of events. In some simulations, such as for plotting fractal algorithms, it is the end result that is important. This type of process is normally carried out as quickly as possible.

More commonly, however, simulations are executed at some regular process rate. Ideally, this rate is fast enough to prevent boredom but slow enough to permit us to observe how the simulation is developing. Generally, for macro simulations such as those in our example, this is compressed time for the simple reason that no one is interested in waiting a year to watch a simulation complete a forest's growth cycle. Alternatively, if we were trying to simulate the first three minutes of the creation of the cosmos, we would probably want to expand time to permit the observation of events that happened too quickly for conventional observation.

In the *Forest* demo, two different compressed time rates are used, each controlled by a system timer: one for normal growth and another for forest fires.

Forest Growth Simulation

Initially, forest growth is simulated at one year = one second, with a one-second timer stepping through the growth cycle. Thus, at one-second intervals, the `ID_GROWTH` timer sends a `WM_TIMER` message to the `WndProc` procedure. In response, a number of events are initialized.

```
case WM_TIMER:
    switch( LOWORD( wParam ) )
    {
        case ID_GROWTH:
            wsprintf( szBuff, szCaption, nYears++ );
            SetWindowText( hwnd, szBuff );
            AgeWorld();
            PropagateTrees();
```

The first response is to update the window caption, displaying the current year (the number of cycles since the initial seeding) before calling `AgeWorld` to cycle the forest through a year's growth.

The next step, `PropagateTrees`, calls a subprocedure to seed new areas from existing growth. In this simulation, three factors control the rate of the spread of the forest:

- The age of the existing growth (within each plot)
- The selection of a single plot within a range of three plots in any direction (quite arbitrarily)
- Whether the target plot is fertile ground (in this case, any bare soil)

All of these factors could be variables or could be changed arbitrarily to experiment with new environmental conditions. Still, the present settings serve as a good foundation for a forest simulation.

Growth and seeding operations are carried out by writing new values to a copy of the original data array, ensuring that the new conditions do not overwrite the existing conditions used to generate the new ones. However, because of this separation of present and future, the `StepForest` procedure is called to copy this future status back to the present array.

In the current simulation, the potential conflicts between the present and future states of the forest are minimal. The second array could be disposed of, and all operations could be carried out in a single array. In other simulations, however, duplicate arrays may be essential. Moreover, circumstances may require several arrays to store not only present and future, but also various data types, which may include constants (such as terrain) or contain factors affecting larger areas (such as rainfall).

Forest Fire Simulation

After the current state of the forest is updated, the `InitBurns` procedure is called to simulate potential forest fires:

```
StepForest();
InitBurns( hwnd );
InvalidateRect( hwnd, NULL, FALSE );
break;
```

When the forest fires (if any) conclude, the display is updated to show the new conditions. The `InitBurns` procedure begins by selecting an arbitrary location for a fire to start:

```
void InitBurns( HWND hwnd )
{
    int x, y, NoChance = 20;
```



```
x = random( GRID );
y = random( GRID );
```

Here, a couple of quite arbitrary conditions have been established:

- Only one fire can be started in any year.
- Any fire that does start completely burns out the affected plot (no plots are partially burned).
- The constant, `NoChance`, is included to adjust the chances for a burn to start.

As with the growth-simulation conditions, all of these factors, including the algorithm, can be changed or may include variables to adjust for various ignition potentials.

To actually decide if a fire starts in the targeted area, a simple algorithm is applied:

```
if( random( iOldGrowth + NoChance ) <= (int) Acres[x][y] )
```

Here, a random value is generated and must be less than the growth state of the plot selected before a fire is initiated. In effect, the older the growth is on the target plot—and therefore the more fuel that is available—the greater the chances are of a fire starting.

Assuming that a fire is initiated, another simulation sequence is started, beginning by setting `nDays` to zero and selecting an initial wind direction:

```
{
    nDays = 0;
    nWind = random(4);
    Acres[x][y] = iBurning;
    PostMessage( hwnd, WM_COMMAND, IDM_STOPGROWTH, 0L );
    PostMessage( hwnd, WM_COMMAND, IDM_STARTBURNS, 0L );
}
```

Last, to initiate the actual burn simulation sequence, the first timer (`ID_GROWTH`) is turned off and a second timer (`ID_FIRES`) is started.

NOTE

Because two separate timers are involved in the simulation, and because the Start and Stop menu options are intended to operate either of these independently, a set of purely internal message procedures is used to trigger these options indirectly rather than directly.

After the ID_FIRES timer has been initiated, all subsequent WM_TIMER messages carry the ID_FIRES identifier and are used to control the new simulation sequence.

Because the burn events are an important departure from the normal growth pattern, the MessageBeep function is used to call attention to these changes. Also, the global nWind wind direction is allowed to change at random intervals before calling the TrackFire subprocedure:

```

case ID_FIRES:
    MessageBeep( 0 );
    if( ! random(4) ) nWind = random(4);
    if( ! TrackFire() )
    {
        PostMessage( hwnd, WM_COMMAND,
                      IDM_STOPBURNS, 0L );
        PostMessage( hwnd, WM_COMMAND,
                      IDM_STARTGROWTH, 0L );
    }
    InvalidateRect( hwnd, NULL, FALSE );
    break;

default:
    return( DefWindowProc( hwnd, msg,
                          wParam, lParam ) );
}
break;

```

As long as fires continue to burn, the TrackFire subprocedure returns TRUE, and the ID_FIRES timer continues uninterrupted. When there are no remaining fires and a FALSE value is returned, the ID_FIRES timer is killed and the ID_GROWTH timer restarted.

The TrackFire subprocedure accomplishes two tasks:

- It ensures that fires do burn out to embers and then to ashy ground. This is governed by a fairly simple algorithm that allows fires to sustain a long burn initially but, after the fires have been burning, causes later burns to develop swiftly but last only briefly. In effect, after a forest fire is well developed, the fires burn hotter and ignite new areas more easily, but burn out faster.
- It provides a means for fires to spread to new areas. This is affected by three factors. The first two, wind direction and wind speed, are relatively obvious. The third is simply a provision to ensure that young acreage does not catch fire and burn off.

Simulation Design

In the *Forest* demo, the processes involving the growth of a forest have been greatly simplified. For example, no provisions have been made to account for rainfall or, during burn phases, for rains that limit or even extinguish the fire's spread. No provisions have been made for seasonal variations, such as wet springs and dry summers or for longer-term climatic variations. Similarly, it includes only a single generic species of tree, with no other plant life present and with no insect damage or disease damage simulated. Likewise, there are no prevailing winds, rainfall patterns, erosion, or soil fertility variations.

Even so, if all of these factors were included in the simulation, this would still be a very simplified cosmos. The point is that any simulation must be restricted to some degree, if only to allow the computer to handle it in a reasonable time and with reasonable memory requirements.

On the other hand, simplifying the simulation does not mean that the results must be simple. Since the object is to model reality, there is certainly every reason for the results of the simulation to mimic the complexities of reality.

Simplification Choices

The task of simplification is threefold:

- To decide which elements of reality are essential to the simulation
- To develop algorithms that mathematically mimic reality
- To present the results of the computations in a format that will show what is happening

Because the *Forest* demo was designed solely to demonstrate a graphic simulation and to serve as an example, many elements reflecting reality that would also have increased the complexity of the program were omitted. Instead, two principal factors were selected for representation:

- Propagation rates and patterns for the development of the forest
- Fires to destroy older growth and make room for new additional growth

Given these two principal factors, the resulting simulation (in terms of burn-off and regrowth patterns) still mimics the patterns observed in natural forests quite accurately, which is exactly what is desired as an initial objective.

In addition to the propagation and burn patterns, one minor evolutionary provision was added: having older growth die off without burning. In the present simulation, the relatively minor decline of old growth by simple attrition is masked by the larger effect of fires, producing the natural pattern observed in dry-climate forests.

If, however, the burn frequency is decreased or stopped, the old growth decline will become a major rather than a minor effect, producing a pattern typical of wetland forests, where large-scale fires are virtually unknown. And, as you may observe by varying the pattern, such a forest will tend toward a steady-state climate forest, which is quite typical of existing older wetland forests where older growth dominates and younger growth is sparse.

Extending a Simulation

Once a basic simulation is operating with a satisfactory degree of validity—when the operations correspond somewhat faithfully to reality—you can add further extensions to simulate additional factors. The advantage of a simulation is that any of these additional factors (or any combination of additional factors) can be tested, varied, and tested again to observe how changes in various parameters affect the progress of the overall simulation.

Even our simple *Forest* demo could be expanded to include more factors and used to study the effects of various cutting patterns and the effect on the recovery and long-term management of sustained yield for a real forest. You might include factors such as elevation and erosion effects, rainfall, and leftover debris from cutting operations and its effects on the spread of fires. Other factors that might affect the overall health of a forest, such as insect infestation and disease, might simply be ignored as irrelevant during the tests. But then, if some validity is found for considering further factors—perhaps the cutting of debris provides a breeding ground for the insect population—then these factors should also be included in the model. Otherwise, the model may show effects that were not anticipated or it may fail to show effects that were.

How you extend a simulation depends on what you are trying to learn. The real value of a simulation is to reveal how processes occur, how elements interact, or where and how patterns appear within a system.

Simulating Mechanical Systems

Modeling of interactions among living systems is just one of the areas in which simulation is useful. Another fruitful area for simulation design lies in analyzing mechanical systems.

Although CAD systems are commonly thought of as design utilities, simulating how mechanical elements interact is an integral part of any mechanical design process. After all, if a set of gears is going to jam when two mechanically driven arms unexpectedly attempt to pass through the same volume of space, it's considerably cheaper to find the flaw in an electronic simulation than to discover this same surprise after tooling up to begin production—or worse, after building a physical model.

You may be thinking that mechanical modeling could certainly be carried out without any accompanying visual elements. After all, it should be faster to calculate (for example) how two gears mesh than to draw the two gears on screen and to repeatedly redraw them as they turn, right? If the only consideration were the two gears, perhaps this would be true. But what about that movement arm driven by the gearing that, in another few seconds of arc, will attempt to pass through one of several mechanical support members? These supports are static and weren't included in the calculated motions—a small oversight, but a mistake that the real universe is not likely to duplicate.

Instead of attempting to calculate the place where every point belonging to every element (both static and dynamic) may potentially interact with some other point, it is simpler to draw the various elements, redrawing each one as often as necessary. This allows the best processors of all, the human eye and brain, to spot the potential conflicts.

Mechanical simulations should not be limited to the interactions of cams, gears, and cogs. Instead, as currently implemented in some virtual reality simulations, the humans (or other creatures) that are using the machines being designed also become part of the simulation process.

WARNING

If you are interested or active in mechanical simulations involving human users, please remember that human beings, unlike machine parts, do not come in standard sizes. Not all men are 5 foot 11 inches tall, 175 pounds, and not all women are 5 foot 2 inches tall, 125 pounds. Design for the taller, smaller, thinner, and heavier people, too.

Speeding Up Floating-Point Calculations

One problem often encountered in simulations has been slow processing caused by repetitive floating-point calculations. In some cases, floating-point calculations simply cannot be avoided. In these cases, the only solution is a fast CPU. In other cases, however, even when fractional interim values are desired or needed, these can be derived using integer rather than floating-point operations, by the simple expedient of limiting accuracy to what is actually required.

For example, suppose that an algorithm that is called repeatedly (several thousand times per simulation cycle) needs to calculate a radial distance using p . Since the result of the calculation will be cast as an integer and used as an index to a data array, the calculation does not need to be carried out to 10 or 12 decimal places, or even to three or four places. Therefore, instead of using a floating-point value for p , such as 3.14159, the calculation can be carried out using integer operations (which are faster) using the value 402 (equals $p * 128$), and then dividing the result by simply shifting the value seven places to the right.

The bit-shift operation is considerably faster than any decimal division, and the entire process is markedly faster than floating-point operations, while achieving (within limits) the same result.

Simulating Theoretical Systems

Physical realities, whether they are living systems or mechanical constructs, offer their own physical appearances as a basis for graphic simulation. Other systems, however, which may be theoretical or nonphysical, do not offer quite the same convenience; instead, these require imagination and artistry in deciding how to display the information generated by a simulation.

As an example, the operations of a computer chip are often simulated by a computer program, particularly during the design process for a new chip. During design, two quite different elements are taken into consideration: the physical and the electronic layouts for the chip.

Of these, the fabricated layout of the chip is relatively simple: How many circuits can be fabricated within a given area of silicon? The electronic layout, on the other hand, is not only more complex but is also directly affected by the physical layout. In this respect, considerations include the signal path between various

elements and, therefore, the signal time between two components; how the components interact electronically; what leakage currents and capacitive effects must be accounted for; and, far from least, how the various components function cybernetically.

So, how is a system this complex simulated? And how is the simulation shown graphically?

First, no single simulation—graphic or otherwise—will suffice (except possibly for a very simple chip), because of the level of physical complexity in contemporary monolithic integrated circuits. Instead (referring strictly to the physical layout), small portions of the chip might be simulated on the screen. Or, for an overview, color-coded areas might represent repetitive circuit areas or areas dedicated to some specific function.

The electronic functions, however, are not this easily coded. They would probably require several different simulations and displays to handle the various elements. For example, a histogram might be used to show signal-path times for different elements. Remember that with today's high-speed chips, even the paths required for clock pulses can be critical, and more than one engineer has vainly expressed a wish for faster electricity. Still, other elements are even less easily displayed. Thus, in many cases, instead of attempting to show the simulations directly, only the results of simulations are shown.

Computer chip and other electronic circuitry designs are only one area involving nonphysical simulations. There are other valid areas that have even less connection to traditional physical reality. Some are simply constructs of our own observations. For example, consider a simulation of the population growth patterns using the Malthusian equation:

$$P_{n+1} = R * P_n * (1 - P_n)$$

where R represents the growth rate for successive populations (P). Okay, population grows linearly, doesn't it? So, wouldn't a simple line graph be appropriate for this equation?

If the whole suggestion sounds like a loaded question, you're right; it is.

First, after a half-dozen initial steps, the equation given is anything but linear. And, second, this particular simulation will yield results unlike anything you might normally expect. In fact, the Malthusian equation is a member of a loose group of formulas referred to as *strange attractors* because of the curves generated over successive reiterations from what initially appear to be only scattered points.

The equation produces an interesting simulation, and simulations can show interesting results that would not be visible simply by examining long columns of numbers.

TIP

To experiment with the Malthusian equation, one convenient method is to plot successive points (P_n and P_{n+1}) as x- and y-coordinate pairs. After a few thousand generations, the resulting plots will begin to show an interesting curve. Next, by varying the growth rate R (try the range from 2.3 to 3.8 in steps of 0.1 or smaller), a single curve becomes an interesting group of curves, complete with inflections and bifurcation points.

In other cases, such as plotting radiation-intensity patterns from a broadcast antenna or reception sensitivity for a receiver, the results are lobes. In other instances, the results are landscapes ranging from smoothly undulating hills and valleys to fields of jagged peaks and crevasses.

Summary

Regardless of the source of the data plotted or the algorithms used for a simulation, the visual presentation is not simply a gimmick to impress board members and visiting bigwigs, but a very valuable tool to allow the use of our own most sensitive tools: color eyesight, superior image processing, and unequaled pattern recognition. Graphics can aid in the simulation of all types of dynamic systems, including both physical and nonphysical systems. The *Forest* demo discussed in this chapter and included on the CD that accompanies this book is a simple example. To learn more, expand the demo to experiment with various effects.

Metafile Operations

- Advantages of metafiles
- Metafiles written to a memory context
- Metafile playback
- Metafiles written to disk files
- Temporary metafiles
- The structure of metafiles

A *metafile* is a method of storing the operations used to create an image such that they can be replayed to recreate the original image in another window or device context.

Metafiles *per se* are not a means of storing or exchanging images; they provide a means of storing or exchanging a record of GDI function operations in a binary format, which create a specific image. You can replay the operations recorded in a metafile to re-create the original image in much the same fashion as you can replay a CD or tape to re-create a voice talking or a piece of music. Although this may sound interesting, as stated, it does not sound particularly useful. Ergo, what good are metafiles?

As with all simple explanations, the preceding description was both accurate and misleading. We'll begin this supplement by talking about some possible applications of metafiles, and you'll see that they can be useful. Then we'll go into the details of recording and playing back metafiles.

Metafile Uses

One of the principal purposes of metafile operations is to exchange images between applications. You can do this directly, using the clipboard operations described in Supplement 22, "Clipboard Data Transfers," or via file operations, as demonstrated in this supplement.

For example, an accounting program could construct a business graph while recording a metafile of the operations involved, then pass the resulting metafile to a text editor, where the image could be re-created for inclusion in a report. This same metafile can also be "played" directly to the printer, without first creating and copying a bitmap image for the purpose.

Another use for metafiles is to record a calculated graphic (image), permitting it to be re-created or duplicated without the need of repeating the calculations. As an intrinsically calculation-intensive example, consider applying this process to a fractal image. The metafile of the fractal could be replayed in a fraction of the time required for the original calculations.

Another advantage of metafiles lies in the storage space required for them versus image files. For example, a scant 150-byte metafile can easily replace a 3,970-byte image file. Disk storage may not be not a consideration, but even with 56kps

modems as the current standard, image-transmission times are still an important factor in more than a few circumstances.

In some cases, metafiles may be preferred over *device-independent bitmap (DIB)* files. Although DIBs have advantages over conventional image files, metafiles are even less device-dependent and adapt automatically to the device context where they are replayed.

Metafiles are not miracle solutions and will not immediately solve all of your programming problems. But metafiles do offer possibilities. Perhaps this list of uses has already suggested a possibility or two relevant to your own applications. If you don't have any ideas yet, you may by the time you finish reading this supplement.

Recording Metafiles

Because metafile operations are easier to demonstrate than to explain, we'll go through the steps used to record a process for a metafile, using an image and a process that should be familiar to you by now. First, in the *PenDraw6* demo program, we will create a seven-pointed star, then complete the image by enclosing the star with a circle provided by the `Ellipse` function.

Creating the Metafile Device Context

The initial requirement prior to recording this metafile is to calculate a series of points describing the seven-pointed star.

```
for( i=j=0; i<7; i++, j=(j+3)%7 )
{
    pt[i].x = (int)( sin( j*PI2/7 ) * 100 );
    pt[i].y = (int)( cos( j*PI2/7 ) * 100 );
}
hdc = BeginPaint( hwnd, &ps );
hPen = CreatePen( PS_NULL, 1, 0L );
SelectObject( hdc, hPen );
SelectObject( hdc, GetStockObject( LTGRAY_BRUSH ) );
Ellipse( hdcMeta, -100, -100, 100, 100 );
```

After calculating the points, creating and selecting a null pen and a standard brush should be familiar operations. With the enclosing circle drawn first, using the null pen and filled by the light-gray brush, the background portion of the image is complete.

Next, still using the null pen, we swap the brush for a dark gray, select the fill mode, and call the `Polygon` function to draw the star inside the light-gray disk.

```
SelectObject( hdc, GetStockObject( DKGRAY_BRUSH ) );
SetPolyFillMode( hdc, ALTERNATE );
Polygon( hdc, pt, 7 );
DeleteObject( hPen );
EndPaint( hdc, &ps );
```

Finally, we delete the null pen.

Executing this same operation to record the process for a metafile is not much different but requires two new variables:

```
static HANDLE hMetaFile;
HDC          hdc, hdcMeta;
```

The two new variables are a static handle, `hMetaFile`, and a device context handle, `hdcMeta`. The declaration of this latter variable, `hdcMeta`, may have already suggested a major element of the changes necessary: the substitution of the `hdcMeta` device context for the more usual `hdc` context. But simply changing the device context ID isn't enough. A more important change appears immediately following:

```
for( i=j=0; i<7; i++, j=(j+3)%7 )
{
    pt[i].x = (int)( sin( j*PI2/7 ) * 100 );
    pt[i].y = (int)( cos( j*PI2/7 ) * 100 );
}
hdcMeta = CreateMetaFile( NULL );
```

Replacing the familiar `BeginPaint` (or `GetDC`) instruction, the `CreateMetaFile` API function provides the metafile equivalent and, in similar fashion, returns a device-context handle. The big difference is that this handle is directed to a device context that is not associated with any physical device. At the same time, since the single parameter has been specified as null, the metafile created will exist in memory only; that is, as a temporary memory file.

Later in this supplement, you'll see another form in which the metafile data is written to a physical (disk) file. But even limited to a memory context, the metafile can still be written to and replayed.

Once the metafile device context has been created, the `hdcMeta` handle can be substituted for the earlier `hdc` handle in the drawing instructions, which now appear thus:

```
hPen = CreatePen( PS_NULL, 1, 0L );
SelectObject( hdcMeta, hPen );
SelectObject( hdcMeta, GetStockObject( LTGRAY_BRUSH ) );
Ellipse( hdcMeta, -100, -100, 100, 100 );
SelectObject( hdcMeta, GetStockObject( DKGRAY_BRUSH ) );
SetPolyFillMode( hdcMeta, ALTERNATE );
Polygon( hdcMeta, pt, 7 );
```

Except for the change in the device context, the drawing operations are precisely the same as those previously directed to the screen context. The image itself, however, has not been drawn to the screen; instead, only the GDI operations necessary to draw the image have been recorded.

Closing and Disposing of the Metafile

Rather than using the `EndPaint` (or `ReleaseDC`) instruction when you are finished using the device context, you use the `CloseMetaFile` instruction:

```
hMetaFile = CloseMetaFile( hdcMeta );
DeleteObject( hPen );
```

The call to the `CloseMetaFile` instruction, unlike an `EndPaint` or `ReleaseDC` instruction, returns a handle not to a device context, but to the metafile itself (which is currently in memory). At this point, this metafile handle can be used to replay the same GDI instructions just calculated.

But unlike the original image, which would have been drawn to match the display context, the GDI instructions can be played back to any device context and will create an image appropriate to the device context.

Last, even though the metafile has been created only in memory, it is still a logical object, and as such, must be disposed of when no longer required (or at the

very least, before the application closes). In this case, the appropriate point is in response to the `WM_DESTROY` message:

```
case WM_DESTROY:
    DeleteMetaFile( hMetaFile );
    ...
```

Replaying Metafiles

Initially, the drawing instructions for our sample image were presented in the form that would have been used to draw to the screen, nominally in response to a `WM_PAINT` instruction. But because these were intended for a metafile rather than a display, the instructions were executed in response to the `WM_INITIALIZE` instruction (see the discussion of the *PenDraw6* demo, later in this supplement). But now that it's time to replay the metafile instructions, this operation will be carried out in response to a `WM_PAINT` operation, in the same fashion as any other screen refresh. However, before actually replaying the metafile, we need to add a few instructions.

Providing a Mapping Mode and Extents

Because the metafile image was drawn to a memory context as GDI instructions, it lacks any physical device-context information, including mapping modes and viewport and window extents. Therefore, before replaying the metafile instructions, we need to provide a physical device-context handle, as well as mapping mode and extent settings:

```
case WM_PAINT:
    hdc = BeginPaint( hwnd, &ps );
    SetMapMode( hdc, MM_ANISOTROPIC );
    SetWindowExt( hdc, 1000, 1000 );
    SetViewportExt( hdc, cxWnd, cyWnd );
```

The fact that the metafile does not include mapping mode and extent settings is an advantage, not a disadvantage. Because these are not predefined within the metafile, before you play back a metafile, you can establish any mapping mode or window and viewport extents desired, and the metafile's GDI operations will be executed accordingly.

Controlling the Image Position

Of necessity, one element is predetermined: the origin point. When the graphics drawing operations were originally executed, the drawing was centered around a (hypothetical) 0,0 origin point, simply because it was convenient. Alternatively, the origin could have been located anywhere in the metaspace, and the resulting operations would be recorded at points relative to this new theoretical origin.

But, while the metafile's origin is known, the viewport and window extents and origins are still undetermined. Because of these two factors, you can control the position of the resulting image by changing the window origin when the metafile image is replayed.

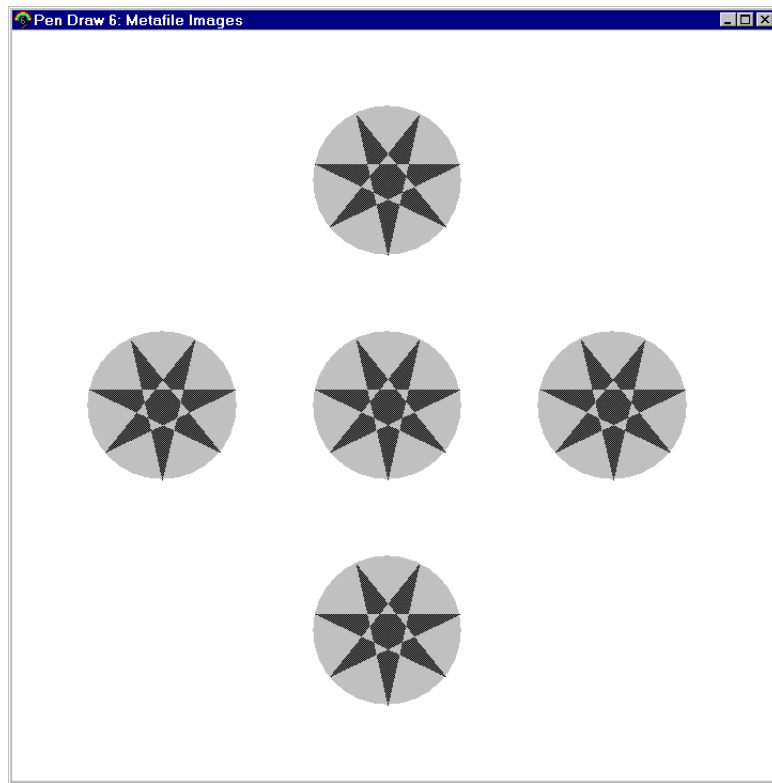
For the present demonstration, we'll replay the metafile image a total of six times, changing the window origin point each time, to produce an image similar to the screen shown in Figure S21.1. (Because two of the resulting images use the same screen coordinates, only five images appear on the screen.)

```
for( i=0; i<3; i++ )
{
    SetWindowOrg( hdc, -200 -( i * 300 ), -500 );
    PlayMetaFile( hdc, hMetaFile );
    SetWindowOrg( hdc, -500, -200 -( i * 300 ) );
    PlayMetaFile( hdc, hMetaFile );
}
EndPaint( hwnd, &ps );
break;
```

In addition to the metafile drawing operations, other drawing operations could also be carried out, either before or after the metafile is replayed. Also, remember that this is not simply an image being copied to the screen; these images are drawn in the same fashion as any other object created by drawing instructions. You should also realize that the metafile may include ROP instructions governing how the recorded drawing operations will interact with existing screen images. (See *Supplement 12* on the CD accompanying this book for more information about ROP instructions.)

FIGURE S21.1:

Five images produced by a metafile



Metafiles as Disk Files

The preceding fragmentary examples for both recording and playing back a metafile assume that the metafile exists only as a memory file. A memory metafile provides an acceptable format for transfer using clipboard functions. For other purposes, you may want to create a disk metafile, by writing the GDI operation instructions to an external disk file in a condensed binary format.

Writing Metafiles to Disk Files

Writing metafiles to disk requires only a minor change in format and could be accomplished, using the preceding examples, by changing one line in the source code:

```
hdcMeta = CreateMetaFile( "D:\\\\METAFILE.WMF" );
```

Neither the filename nor the file extension used here has any particular significance, although the .WMF extension (Windows MetaFile) does provide a convenient convention.

The `CreateMetaFile` instruction could also be written using an indirect reference:

```
hdcMeta = CreateMetaFile( (LPSTR) szMetaFileName );
```

In this form, `szMetaFileName` is a null-terminated string specifying the filename and, if desired, the drive and path specifications.

In either case, when the metafile is written to disk, the `DeleteMetaFile` instruction in response to the `WM_DESTROY` message (shown earlier in the supplement) does not affect the metafile disk file. The `DeleteMetaFile` instruction deletes only the local handle to the file. The file itself remains until explicitly erased by other instructions.

As another alternative, you can create a temporary file. This form of storage is more ephemeral than a conventional disk file, but less transitory than a memory file. This is the format used in the *PenDraw6* demo discussed in this supplement.

Generating Temporary Files

To create a temporary file, the demo uses the `GetTempFileName` function to create a temporary filename and calls it in response to the `WM_CREATE` function, instead of supplying a string constant when calling the `CreateMetaFile` function. The `GetTempFileName` function is called as:

```
GetTempFileName( lpszDrivePath, lpszPrefixStr,  
                wUnique, lpszTempFileName );
```

The function's parameters are as follows:

lpszDrivePath This parameter points to a null-terminated string specifying the drive and directory where the temporary file will be located. If no drive or path is specified, the current drive and directory will be used. (If desired, a call to `GetTempPath` will return the path name of the system's predefined temporary file directory.)

lpszPrefixStr This parameter points to a null-terminated string to be used as the prefix for a temporary filename. It is limited to three characters in length.

wUnique This is an unsigned short integer (WORD) used to generate the temporary filename. If this argument is zero, a unique number (based on the system time) will be generated and will also be returned by `GetTempFileName`. If a file already exists with that generated name, the number will be incremented and the new name tested for conflicts, continuing until a unique filename is found.

lpszTempFileName This parameter points to a buffer that receives the temporary filename. The return string consists of characters in the OEM-defined character set and should be at least `MAX_PATH` (260) characters in length to allow sufficient room for a complete drive/path/filename specification.

In actual practice (and in the *PenDraw6* demo), `GetTempFileName` can also be called as:

```
GetTempFileName( NULL, "MFT", 0, (LPSTR) szMetaFileName );
```

The *PenDraw6* demo includes a message box provision that reports the temporary filename generated. As you will observe, the temporary filename created begins with a tilde character (~), followed by the optional prefix ("MFT") and completed with a unique four-character hexadecimal value generated from the `wUnique` parameter. For example, the returned drive/path/filename might be `C:\\Win2000\\TEMP\\~MFT12F3.TMP`.

TIP

Notice that the path specification uses double backslash characters, as required by C-language conventions.

Deleting Temporary Files

Because large numbers of temporary files tend to clog a hard drive (a fault that, unfortunately, is characteristic of too many existing Windows applications), any applications using temporary files should also include provisions to erase these files when they are no longer needed.

This task can be accomplished quite easily by minor provisions in response to the `WM_DESTROY` message.

```
case WM_DESTROY:
    DeleteMetaFile( hMetaFile );
    unlink( szMetaFileName );
    PostQuitMessage(0);
    break;
```

The `unlink` function erases the temporary file without requiring a file handle or other handling provisions. (Remember that read-only files cannot be unlinked.)

There is one flaw in this system: If an application is interrupted—because of a hang-up requiring a reset, a power interruption, or any other reason—any temporary files created will not be erased, and you will need to manually delete them from the hard drive. Of course, this can also be done to clean up after other, less well-behaved applications, and this is also a good reason for using the `GetTempPath` function. With this last provision, all garbage files will remain in one convenient location where they're easily found and deleted—a small courtesy, but an appreciated one.

Accessing Temporary Metafiles

The principal reason for creating temporary metafiles is to allow other applications to access the information that they contain (an application also might need to access metafiles that it created earlier, but a temporary file format would probably not be used in this case). Thus, we need a provision to retrieve or create a metafile handle for a file that was not created by the application or that was previously discarded via the `DeleteMetaFile` function. This facility is provided by the `GetMetaFile` function, which is called as:

```
hMetaFile = GetMetaFile( (LPSTR) szMetaFileName );
```

Once the handle has been retrieved, we can access the metafile as before. When finished, we should discard the metafile handle using `DeleteMetaFile`.

If the metafile is being created by one application for use by another, the originating application should not call `unlink` to delete the disk file, but the recipient application definitely should.

Metafile Structures

Metafiles, whether in memory or on disk, are simply structured records using the `METARECORD` and `METAHEADER` structures, or using the `METAFILEPICT` structure for clipboard operations, which is discussed in Supplement 22.

You do not need to know how to read or decipher metafile instructions in order to use metafile operations. However, being able to do so may prove worthwhile, and they really aren't as difficult to understand as you might assume.

A metafile begins with an 18-byte record header described by the `METAHEADER` structure. This structure is followed by a series of `METARECORD` records, each consisting of a minimum of four `WORD` values, which describe the actual GDI operations.

The Metafile Header Structure

The `METAHEADER` structure is defined in `WinGDI.H` as:

```
typedef struct tagMETAHEADER
{
    WORD    mtType;           // metafile type
    WORD    mtHeaderSize;     // header size (bytes)
    WORD    mtVersion;        // version number
    DWORD   mtSize;           // metafile size (bytes)
    WORD    mtNoObjects;
    DWORD   mtMaxRecord;
    WORD    mtNoParameters;
} METAHEADER;
```

The Metafile Record Structure

The `METARECORD` structure is defined in `WinGDI.H` as:

```
typedef struct tagMETARECORD
{
    DWORD   rdSize;           // record size
    WORD    rdFunction;       // function ID
    WORD    rdParm[1];
} METARECORD;
```

Each `METARECORD` records a specific GDI function call and varies in length with the first `DWORD` value (`rdSize`) identifying the total size of the individual record.

This value is expressed in *lsw,msw* order, with each word expressed in *lsb,msb* order.

The second record element, *rdFunction*, identifies the function to be executed. The low byte identifies the GDI function, and the high byte reports the number of parameters passed to the function. Thus, a hex value 0418 identifies the *Ellipse* function (18h or *META_ELLIPSE*), which receives four (04h) parameters, excluding the *hdcMeta* argument (which is understood).

Metafile operation constants are defined in *WinGDI.H*. Table S21.1 lists some representative metafile operations.

TABLE S21.1: Representative Metafile Operations

Metafile Operation	Value	Op ID	Arguments
<i>META_SETBKCOLOR</i>	0x0201	01h	2
<i>META_SETBKMODE</i>	0x0102	02h	1
<i>META_SETMAPMODE</i>	0x0103	03h	1
<i>META_SETROP2</i>	0x0104	04h	1
<i>META_SETRELABS</i>	0x0105	05h	1
<i>META_SETPOLYFILLMODE</i>	0x0106	06h	1
<i>META_SETSTRETCHBLTMODE</i>	0x0107	07h	1
<i>META_SETTEXTCHAREXTRA</i>	0x0108	08h	1
<i>META_SETTEXTCOLOR</i>	0x0209	09h	2
<i>META_SETTEXTJUSTIFICATION</i>	0x020A	0Ah	2
<i>META_SETWINDOWORG</i>	0x020B	0Bh	2
<i>META_SETWINDOWEXT</i>	0x020C	0Ch	2
<i>META_SETVIEWPORTORG</i>	0x020D	0Dh	2
<i>META_SETVIEWPORTEXT</i>	0x020E	0Eh	2
<i>META_OFFSETWINDOWORG</i>	0x020F	0Fh	2
<i>META_SCALEWINDOWEXT</i>	0x0410	10h	4

Continued on next page

TABLE S21.1 CONTINUED: Representative Metafile Operations

Metafile Operation	Value	Op ID	Arguments
META_OFFSETVIEWPORTORG	0x0211	11h	2
META_SCALEVIEWPORTEXT	0x0412	12h	4
META_LINETO	0x0213	13h	2
META_MOVETO	0x0214	14h	2
META_EXCLUDECLIPRECT	0x0415	15h	4
META_INTERSECTCLIPRECT	0x0416	16h	4
META_ARC	0x0817	17h	8
META_ELLIPSE	0x0418	18h	4
META_FLOODFILL	0x0419	19h	4
META_PIE	0x081A	1Ah	8
META_RECTANGLE	0x041B	1Bh	4
META_ROUNDRECT	0x061C	1Ch	6

The final element in the METARECORD structure will be one or more bytes containing the arguments required by the GDI operation.

Sample Metafile Instructions

Using the metafile operations recorded by the *PenDraw6* demo, as written to a temporary (disk) metafile, a sample of metafile instructions is shown below. The sample code has been interlineated with the program instructions generating each metafile record, and the META_xxxxxxx operation is named at the right. Notice also that, in some cases, a single source code line may generate more than one metafile instruction record.

```

0001 0009 0300 00000004B 0003 00000012 0000  METAHEADER Record

      hPen = CreatePen( PS_NULL, 1, 0L )
00000008 02FA 0005 0001 0000 0000 0000  META_CREATEPENINDIRECT

```

```

    SelectObject( hdcMeta, hPen );
00000004 012D 0000                                META_SELECTOBJECT

    SelectObject( hdcMeta, GetStockObject( LTGRAY_BRUSH ) );
00000007 02FC 0000 C0C0 00C0 0000                META_CREATEBRUSHINDIRECT
    (notice that LTGRAY_BRUSH is expressed as an RGB quad)
00000004 012D 0002                                META_SELECTOBJECT

    Ellipse( hdcMeta, -100, -100, 100, 100 );
00000007 0418 0064 0064 FF9C FF9C                META_ELLIPSE

    SelectObject( hdcMeta, GetStockObject( DKGRAY_BRUSH ) );
00000007 02FC 0000 4040 0040 0000                META_CREATEBRUSHINDIRECT
    (again, DKGRAY_BRUSH is specified as an RGB quad)
00000004 012D 0002                                META_SELECTOBJECT

    SetPolyFillMode( hdcMeta, ALTERNATE );
00000004 0106 0001                                META_SETPOLYFILLMODE

    Polygon( hdcMeta, pt, 7 );
00000012 0324 0007 0000 0064 002B FFA6            META_POLYGON
    FFB2 003E 0061 FFEA FF9F FFEA
    004E 003E FFD5 FFA6
    (includes 14 coordinates-7 points-from pt reference)

    (A 3-byte null record terminates the metafile)
00000003 0000

```

NOTE

If you compare the instructions in the metafile with the complete source code generating these instructions, you may notice a few discrepancies. Where the original instruction was `CreatePen`, the metafile equivalent has become `CreatePenIndirect`. Also, the `DeletePen` instruction in the original source code is not reflected in the metafile instructions. Neither of these are errors, but instead are simplifications of the original code. Because the `CreatePenIndirect` instruction was used instead of `CreatePen`, the need for a `DeletePen` instruction has been eliminated, as has any requirement to restore the original pen. Thus, for the sake of brevity, the metafile translation has improved on the original code.

Again, you do not need to know the structure of a metafile in order to use metafile operations, but the information may prove helpful. Besides, given the preceding breakdown, it really shouldn't be much of a challenge to write a utility

to decipher/decode metafile instructions, should it? If you're interested, you can experiment on your own.

Metafile Cautions

Before leaving the subject of metafile operations, there are a few comments and cautions that are worth keeping in mind. Even a cursory awareness of these may assist in preventing future errors, or at least alleviate confusion.

The metafile device context is not a true device context in the sense that it does not correspond to any physical or logical device. As such, the metafile device context does not include a mapping mode, window sizes and origins, or viewport sizes and origins.

All parameters passed to metafiles are actual values, not formulas or references to variable values. Thus, variable references used in generating an application's source code are evaluated at the time the metafile is compiled and may or may not contain appropriate values when the metafile is replayed. An argument such as `CxWnd/2` is recorded as the constant resulting from the calculation, and will not reflect subsequent changes in window size. (This specific conflict, despite the use of a similar reference, was avoided in the *PenDraw6* demo by using the isotropic mapping mode.)

Metafile instructions are always interpreted in terms of the existing mapping mode. Metafiles may, however, include instructions to select specific mapping modes. Also, there are several classes of instructions that are not compatible with metafile operations. The following five categories of GDI instructions are invalid and will not be recorded as metafile operations:

- Any function treating the metafile device context as if it were a physical device context, including operations such as `CreateCompatibleBitmap`, `CreateCompatibleDC`, `CreateDiscardableBitmap`, `DeleteDC`, `PlayMetaFile` (self-referential), and `ReleaseDC`.
- Any function beginning with the form `Get`, such as `GetDeviceCaps` and `GetTextMetrics`. All data contained in a metafile is preset, and the record structure cannot accommodate information returned by such functions.

- Any function designed to return information to the program, such as `DPTOLP` and `LPtoDP`. (However, macros—which are evaluated during compilation—are permitted.)
- Functions requiring handles to brushes, such as `FillRect` and `FrameRect`.
- A few of the more complex functions, such as `DrawIcon`, `GrayString`, and `SetBrushOrg`.

If you have any questions about which GDI function calls are permitted in a metafile operation, refer to the list of metafile constants defined in `WinGDI.h`. All constants begin with the prefix `META_`. But remember, some GDI functions do not appear, simply because when the compiler encounters these, it will automatically choose a more compatible variation. For example, a call for the `CreatePen` function appears in the metafile as `META_CREATEPENINDIRECT`.

One final caution involves saving the present device context before replaying a metafile and, when finished, restoring the original device context. Because a metafile can change device-context settings but cannot record or restore existing device-context settings, your applications should include their own provisions for this operation.

Remember, the metafile is free to change drawing and mapping modes, change colors, and make other changes. When the metafile is finished replaying, these changes will remain in effect. Therefore, to save the original device context, before the `PlayMetaFile` instruction is executed, save the existing device context:

```
SaveDC( hdc );
```

After the metafile has been replayed, restore the original device context:

```
RestoreDC( hdc, -1 );
```

Since neither of these instructions involves any operations forbidden to metafiles and both are supported by metafile instructions, a well-behaved metafile could simply include these provisions within itself. Do remember, however, that every call to `SaveDC` must have a corresponding `RestoreDC` call using the `-1` argument, and vice versa.

Summary

As you've seen in this supplement, metafile operations provide a powerful means to record and replay powerful drawing operations. They can also be used to transfer graphics operations between applications or even between devices (such as from the screen to a printer). In the next supplement, you'll see how metafiles can be used with the clipboard as an alternative to physical (disk) file transfers. The DDE functions detailed in Supplement 22 provide a means useful for requesting and confirming metafile transfers.

The *PenDraw6* demo on the accompanying CD provides a platform for experimentation with metafile operations. To further your expertise and understanding, you might also consider creating two new programs: the first to create a metafile as a disk file and the second to retrieve the metafile from disk, using the `Get-MetaFile` instruction. Then you can replay the graphics under the same or different mapping modes.

Clipboard Data Transfers

- Advantages of using the clipboard
- Clipboard data formats
- Clipboard access
- Private clipboard formats

The Windows Clipboard consists of two quite different entities: the clipboard viewer, `Clipbrd.EXE`, and the real clipboard. The real clipboard is a feature of the Windows User module. It provides a series of functions that facilitate the temporary storage of information in a form that permits applications other than the originating application to retrieve that information. Of course, the originating application is not prohibited from retrieving its own clipboard information, but the important item to remember is that data passed to the clipboard is public, which means that it is accessible to any application.

The clipboard provides a useful and convenient method for exchanging data of many different types between applications, as explained and demonstrated in this supplement.

Clipboard Uses

The clipboard consists of a series of facilities that provide a platform for the temporary (non-disk) storage of data. Data stored on the clipboard can be transferred between applications or simply retrieved by the application that put it there in the first place.

A source application can copy data to the clipboard using one of the predefined formats or using a custom format (clipboard file formats are discussed later in this supplement). As the data is transferred, the clipboard facilities allocate and manage memory to contain the data. After the data has been transferred, any application can access the clipboard, inquire what type of data is present, and, if desired, retrieve a copy of the data from the clipboard.

When the clipboard viewer (`Clipbrd.EXE`) is active, the clipboard is queried regularly to determine if any data has been written to the clipboard and, if so, what data type is contained. If possible, the clipboard viewer then retrieves a copy of the data, displaying the data in its own client window. Note, however, that the clipboard viewer itself does not alter or erase the clipboard contents.

Although the clipboard does work very well, it also has a few disadvantages:

There is only one clipboard. Because there is only one clipboard, all applications that want to use the clipboard must share use of this single facility. And sharing can mean conflicts. For example, suppose that Application A writes a bitmap to the clipboard and then Application B writes a block of

text data. However, because Application B, quite reasonably begins by clearing the clipboard, the bitmap written by Application A is erased. Now, if the bitmap destination, Application C, has already retrieved the image, everything is fine. But, if Application C has not gotten the bitmap before Application B replaces it with text data, the bitmap is lost.

All material written to the clipboard is public. The public nature of the clipboard also offers opportunities for error. Because the data element written to the clipboard cannot be addressed to a specific recipient, this data can be accessed, by mistake, by another application seeking data of the same type.

Any material written to the clipboard is volatile. The clipboard can contain data of several different types, written by a single application or by different applications. If this is the case, the problem is how to distinguish between the blocks—for example, multiple blocks of text supplied by different sources. For this reason, applications normally begin by clearing the clipboard before writing new material to the clipboard.

These are factors to consider, but they are not serious problems demanding extensive worry and circumvention measures. And, in circumstances where these could become more serious considerations, other processes such as named or anonymous pipes (see Chapter 7, “Processes and Pipes”) or memory-mapped files (see Chapter 10, “Memory Management”) provide more secure channels for the exchange of data. Alternately, you can also consider using OLE client/server transactions (see Supplement 23, “OLE Client/OLE Server”) or COM services (see Chapter 23, “New COM Features in Windows 2000”).

The Clipboard Viewer

The `Clipbrd.EXE` program, which is distributed with all versions of Windows, is a clipboard viewer that provides a means of checking (viewing) data that has been copied to the User clipboard facilities. As such, the `Clipbrd` program can be used while testing your own clipboard routines.

The clipboard viewer can also be used to capture (or copy) material transferred to the clipboard facilities by other applications, saving the captured data to a disk file or simply viewing the data.

But, remember, the `Clipbrd` application is only a viewer, not the real clipboard. It cannot affect the contents of the clipboard.

Clipboard Operations

Basically, the clipboard operates by assuming control over globally allocated memory blocks that contain data supplied by applications, by altering memory allocation flags. To copy or write material to the clipboard, an application begins by using the `GlobalAlloc` function and the `GHND` flag (defined as `GMEM_MOVABLE` and `GMEM_ZEROINIT`) to initialize a memory block that initially belongs to the originating application instance.

Under normal circumstances, when the originating application exits or closes, the global memory allocated would be deleted (freed) by Windows. However, when the originating instance calls the `SetClipboardData` function, using the global handle to the memory block, Windows transfers ownership of the memory block from the application to itself to the clipboard by modifying the memory allocation flags for the global memory block.

Ownership of a global memory block is accomplished by the `GlobalReAlloc` function, called as:

```
GlobalReAlloc( hMem, NULL, GMEM_MODIFY | GMEM_DDESHARE );
```

Once this is done, the allocated memory no longer belongs to the original application and can now be accessed only through the clipboard using the `GetClipboardData` function. The `GetClipboardData` function grants the calling application temporary access to the clipboard data by providing a handle to the global memory block. However, ownership remains with the clipboard, not with the application accessing the data.

For this reason, clipboard data can be erased only by calling the `EmptyClipboard` function. (One exception to this rule will be discussed later, but is not recommended.)

Clipboard Data Formats

Windows supports two dozen standard clipboard data formats, each identified by enumerated values defined in `WinUser.H`. Of these two dozen formats, these fourteen are probably the most important:

<code>CF_TEXT</code>	<code>CF_PALETTE</code>
<code>CF_BITMAP</code>	<code>CF_PENDATA</code>
<code>CF_METAFILEPICT</code>	<code>CF_RIFF</code>

CF_SYLK	CF_WAVE
CF_DIF	CF_DIB
CF_TIFF	CF_UNICODETEXT
CF_OEMTEXT	CF_ENHMETAFILE

In addition to the predefined formats (listed in Table S22.1) any application is free to define its own custom clipboard data format.

TABLE S22.1: Predefined Clipboard Formats

Value	Meaning
CF_BITMAP	A handle to a bitmap (HBITMAP).
CF_DIB	A memory object containing a BITMAPINFO structure followed by the bitmap bits.
CF_DIBV5	Windows 2000 (NT 5.0): A memory object containing a BITMAPV5HEADER structure followed by the bitmap color space information and the bitmap bits.
CF_DIF	Software Arts' Data Interchange Format.
CF_DSPBITMAP	Bitmap display format associated with a private format. The <i>hMem</i> parameter must be a handle to data that can be displayed in bitmap format in lieu of the privately formatted data.
CF_DSPENHMETAFILE	Enhanced metafile display format associated with a private format. The <i>hMem</i> parameter must be a handle to data that can be displayed in enhanced metafile format in lieu of the privately formatted data.
CF_DSPMETAFILEPICT	Metafile picture display format associated with a private format. The <i>hMem</i> parameter must be a handle of data that can be displayed in metafile picture format in lieu of the privately formatted data.
CF_DSPTTEXT	Text display format associated with a private format. The <i>hMem</i> parameter must be a handle of data that can be displayed in text format in lieu of the privately formatted data.
CF_ENHMETAFILE	A handle (HENHMETAFILE) of an enhanced metafile.
CF_GDIOBJFIRST ... CF_GDIOBJLAST	Range of integer values for application-defined GDI object clipboard formats. Handles associated with clipboard formats in this range are not automatically deleted using the <code>GlobalFree</code> function when the clipboard is emptied. Also, when using values in this range, the <i>hMem</i> parameter is not a handle to a GDI object, but is a handle allocated by the <code>GlobalAlloc</code> function with the <code>GMEM_DDESHARE</code> and <code>GMEM_MOVEABLE</code> flags.

Continued on next page

TABLE S22.1 CONTINUED: Predefined Clipboard Formats

Value	Meaning
CF_HDROP	A handle of type <code>HDROP</code> that identifies a list of files. An application can retrieve information about the files by passing the handle to the <code>DragQueryFile</code> functions.
CF_LOCALE	<p>The data is a handle to the locale identifier associated with text in the clipboard. When closing a clipboard containing <code>CF_TEXT</code> data but no <code>CF_LOCALE</code> data, the system automatically sets the <code>CF_LOCALE</code> format to the current input locale. Alternately, a <code>CF_LOCALE</code> format for a different locale can be associated with the clipboard text.</p> <p>An application pasting text from the clipboard can retrieve this locality format to determine which character set was used to generate the text. (Note that the clipboard does not support plain text for multiple character sets; instead, a formatted text data type such as RTF can be used.)</p> <p>Windows NT/2000: The system uses the code page associated with <code>CF_LOCALE</code> to implicitly convert from <code>CF_TEXT</code> to <code>CF_UNICODETEXT</code>, ensuring that the correct code page table is used for the conversion.</p>
CF_METAFILEPICT	Handle of a metafile picture format as defined by the <code>METAFILEPICT</code> structure. When passing a <code>CF_METAFILEPICT</code> handle by means of dynamic data exchange (DDE), the application responsible for deleting <i>hMem</i> should also free the metafile referred to by the <code>CF_METAFILEPICT</code> handle.
CF_OEMTEXT	Text format containing characters in the OEM character set. Each line ends with a carriage return/linefeed (CR/LF) combination. A null character signals the end of the data.
CF_OWNERDISPLAY	Owner-display format. The clipboard owner must display and update the clipboard viewer window, and receive the <code>WM_ASKCBFORMATNAME</code> , <code>WM_HSCROLLCLIPBOARD</code> , <code>WM_PAINTCLIPBOARD</code> , <code>WM_SIZECLIPBOARD</code> , and <code>WM_VSCROLLCLIPBOARD</code> messages. The <i>hMem</i> parameter must be <code>NULL</code> .
CF_PALETTE	<p>Handle of a color palette. Whenever an application places data in the clipboard that depends on or assumes a color palette, it should place the palette on the clipboard as well.</p> <p>If the clipboard contains data in the <code>CF_PALETTE</code> (logical color palette) format, the application should use the <code>SelectPalette</code> and <code>RealizePalette</code> functions to realize (compare) any other data in the clipboard against that logical palette.</p> <p>When displaying clipboard data, the clipboard always uses as its current palette any object on the clipboard that is in the <code>CF_PALETTE</code> format.</p>
CF_PENDATA	Data for the pen extensions to Microsoft Windows for Pen Computing.

Continued on next page

TABLE S22.1 CONTINUED: Predefined Clipboard Formats

Value	Meaning
CF_PRIVATEFIRST ... CF_PRIVATELAST	Range of integer values for private clipboard formats. Handles associated with private clipboard formats are not freed automatically; the clipboard owner must free such handles, typically in response to the WM_DESTROYCLIPBOARD message.
CF_RIFF	Represents audio data more complex than can be represented in a CF_WAVE standard wave format.
CF_SYLK	Microsoft Symbolic Link (SYLK) format.
CF_TEXT	Text format. Each line ends with a carriage return/linefeed (CR/LF) combination. A null character signals the end of the data. Use this format for ANSI text.
CF_TIFF	Tagged-image file format.
CF_UNICODETEXT	Windows NT/2000: Unicode text format. Each line ends with a carriage return/linefeed (CR/LF) combination. A null character signals the end of the data.
CF_WAVE	Represents audio data in one of the standard wave formats, such as 11kHz or 22kHz pulse code modulation (PCM).

Text Formats

The simplest clipboard data format is the CF_TEXT format, which consists of null-terminated ANSI character strings, each line ending with a carriage return (0x0D)/line feed (0x0A) character. The CF_OEMTEXT format is an OEM character set. The CF_UNICODETEXT format uses 32-bit Unicode characters.

Once the text has been transferred to the clipboard, the originating application cannot access the text further except by requesting access from the clipboard.

Bitmap Format

The CF_BITMAP format is used to transfer Windows bitmap images by transferring the bitmap handle to the clipboard. Once the bitmap handle has been transferred to the clipboard, the originating application cannot use the bitmap except by calling the clipboard for access.

Metafile Formats

The CF_METAFILEPICT format is used to transfer memory (not disk) metafiles between applications. This format uses the METAFILEPICT structure, defined in `WinGDI.H` as:

```
typedef struct tagMETAFILEPICT
{
    LONG        mm;
    LONG        xExt;
    LONG        yExt;
    HMETAFILE    hMF;
} METAFILEPICT, FAR *LPMETAFILEPICT;
```

The first three fields show the differences between a clipboard metafile transfer and a disk metafile transfer. The first field, `mm`, identifies the preferred mapping mode (discussed later). The second and third fields, `xExt` and `yExt`, identify the height and width of the metafile image. The `HMETAFILE` field is simply a handle to the `METAFILE` structure introduced in Supplement 21, “Metafile Operations.” The use of this data is demonstrated later in this chapter.

The `CF_ENHMETAFILE` format is the same as the `CF_METAFILEPICT` format, except that it identifies a metafile using the enhanced metafile format instructions.

Once a metafile is transferred to the clipboard, the originating application should not attempt to use either the global memory block or the original metafile handle, except by requesting access through the clipboard.

DIB Format

The `CF_DIB` format is used to transfer device-independent bitmaps (DIBs) to the clipboard. Each DIB is transferred as a global memory block, beginning with a `BITMAPINFO` header structure, followed by the bitmap image data. (Bitmap image structures are discussed in Supplement 13 on the CD.)

NOTE

The `CF_BITMAP` format supported by Windows 2.x and 3.x identifies device-dependent bitmap formats that are also supported by Windows 98 (but not by Windows NT/2000). However, the `CF_DIB` format is preferred, and is supported by all Windows versions from 95 through 2000.

After a bitmap has been transferred to the clipboard, the originating application should not attempt to use either the global memory block or the original bitmap handle, except by requesting access through the clipboard.

Palette and Pen Formats

The CF_PALETTE and CF_PENDATA formats are used to transfer a handle to a color palette or a pen, respectively. The palette transfer is often used together with the CF_DIB format to define color palettes used by a bitmap.

Wave Format

The CF_WAVE format is used to transfer audio (waveform) information between applications.

Special-Purpose Formats

Three special-purpose clipboard formats provide support for data formats that were originally designed for use by and between specific applications:

CF_TIFF Uses a global memory block to transfer data using the Tagged Image File Format (TIFF). (See Supplement 15 for more information about TIFF files.)

CF_DIF Uses a global memory block to transfer data using the Data Interchange Format (DIF) created by Software Arts, originally for use with the VisiCalc spreadsheet program but now controlled by Lotus Corporation. The format is essentially an ASCII-string format with each line terminated by CR/LF pairs.

CF_SYLK Uses a global memory block to transfer data using the Microsoft Symbolic Link format, originally designed for data exchanges between Microsoft's Multiplan (spreadsheet), Chart, and Excel applications. The format is an ASCII string format with each line terminated by a CR/LF pair.

Accessing the Clipboard

While many Windows facilities are designed to permit shared access, access to the clipboard is permitted to only one application at a time. This restriction prevents conflicts among applications.

Opening and Closing the Clipboard

Before any application can access the clipboard to read, write, or clear it, the application must begin by calling the `OpenClipboard` function to request access. The `OpenClipboard` function returns a Boolean result, with `TRUE` indicating that the clipboard is available and access is granted or `FALSE` indicating that access is denied because another application currently holds access rights.

When an application is finished with the clipboard, the `CloseClipboard` function is called, relinquishing access and freeing the clipboard for access by other applications.

WARNING

Remember that the `OpenClipboard` function must always be matched with a `CloseClipboard` call. Emphasis on the *always*! An application should never, ever attempt to hold the clipboard open, and should always relinquish control of the clipboard as quickly as possible.

Transferring Data to the Clipboard

The *ClipBoard* demo discussed in this chapter provides an example of a clipboard transfer function that copies a memory block to the clipboard.

```
BOOL TransferToClipboard( HWND hwnd, HANDLE hMemBlock,
                          WORD FormatCB )
{
    if( OpenClipboard( hwnd ) )
    {
        EmptyClipboard();
        SetClipboardData( FormatCB, hMemBlock );
        CloseClipboard();
        return( TRUE );
    }
    return( FALSE );
}
```

The `TransferToClipboard` function begins by requesting access (opening) the clipboard, then copying a single memory block to the clipboard. Last, the clipboard is closed, relinquishing further access.

The `TransferToClipboard` function is quite generic in design, accepting any type of handle (`hMemBlock`). However, it does require the `FormatCB` parameter to specify the format type (the type of data copied to the clipboard).

The term *memory block* does not refer to a specific size; the size of the memory block was set earlier by the `GlobalAlloc` function. A single memory block might contain paragraphs of text, multiple records, or any other data. Each memory block, however, can contain only one data type.

So, what if an application needs to transfer a bitmap, a metafile, a palette, and a text block? The solution is relatively simple. First, each block is copied, separately, to globally allocated memory, retaining a handle to each memory block (in the following example, `hBitmap`, `hPalette`, `hMetafile`, and `hText`). With this done, the clipboard is opened and emptied, then each of the handles is transferred to the clipboard:

```
if( OpenClipboard( hwnd ) )
{
    EmptyClipboard();
    SetClipboardData( CF_BITMAP, hBitmap );
    SetClipboardData( CF_PALETTE, hPalette );
    SetClipboardData( CF_METAFILEPICT, hMetafile );
    SetClipboardData( CF_TEXT, hText );
    CloseClipboard();
}
```

Last, the clipboard is closed, relinquishing further access to other applications.

In actual practice, the preceding example would be rather cumbersome; providing source code for every possible combination of data types would be more than a little frustrating. But the data type identifiers are all `WORD` values, and the memory block handles are simply that—handles. A simpler form would be to begin by assigning the data types and handle as arrays of `WORD` and `HANDLE`, and then calling the transfer function with a further parameter reporting the number of items to transfer. This done, the transfer could be handled in this way:

```
if( OpenClipboard( hwnd ) )
{
    EmptyClipboard();
    for( i=0; i<nCount; i++ )
        SetClipboardData( cfType[i], hData[i] );
    CloseClipboard();
}
```

Retrieving Clipboard Data

Before attempting to retrieve an item from the clipboard, the first step is to find out if the clipboard holds a particular type of data. Because different data types require different handling after they are retrieved, applications need to know what they're retrieving and be prepared to handle the result before requesting retrieval.

One method is to simply ask for data of a desired type and see if anything is returned. But this approach does lack a certain elegance, not to mention efficiency. The more efficient way to find out about the clipboard contents is to use one of the two supplied functions: `IsClipboardFormatAvailable` or `EnumClipboardFormats`. The `IsClipboardFormatAvailable` function returns a Boolean result to report if the clipboard contains a desired data format.

`IsClipboardFormatAvailable` is called as:

```
if( IsClipboardFormatAvailable( CF_xtypexx ) ) ...
```

The `EnumClipboardFormats` function queries all available clipboard formats. When you initially call `EnumClipboardFormats` with a NULL parameter, it reports the first available format. Each time you call the `EnumClipboardFormats` function, it returns a value reporting the next available format. Thus, to request a list of all available formats:

```
wFormat = NULL;  
OpenClipboard( hwnd );  
while( wFormat = EnumClipboardFormats( wFormat ) )  
{  
    ... code handling various formats ...  
}  
CloseClipboard();
```

The formats returned are reported in the same order as the originating application used to paste items to the clipboard. This ordering allows the querying application to respond to the first format acceptable. The originating application can post items in a recommended order—for example, in order of descending data reliability.

If no further formats are available, if the clipboard is empty, or if the clipboard has not been opened, the return result will be zero. The `wFormat` parameter could be reset to a specific value to repeat the list from that point. Also, the number of formats available in the clipboard can be retrieved by calling:

```
nFormats = CountClipboardFormats();
```

Once an application has determined that the clipboard contains data of a desired type, retrieving the clipboard data consists of two operations:

- Retrieving a handle to the clipboard data, the memory block
- Doing something with the data after retrieving the handle

The first is quite simple, as illustrated by the `RetrieveCB` function:

```
HANDLE RetrieveCB( HWND hwnd, WORD FormatCB )
{
    HANDLE hCB;

    if( ! IsClipboardFormatAvailable( FormatCB ) )
        return( NULL );
    OpenClipboard( hwnd );
    hCB = GetClipboardData( FormatCB );
    CloseClipboard();
    return( hCB );
}
```

This example offers a generic subroutine that returns an untyped handle to a clipboard memory block. If the requested type is not available, it returns `NULL`.

In actual practice, a slightly different format is used, as in the *ClipBoard* demo where a request is made to the clipboard for a metafile object:

```
nClipRetrieve = 0;
if( IsClipboardFormatAvailable( CF_METAFILEPICT ) )
{
    OpenClipboard( hwnd );
    hGMem = GetClipboardData( CF_METAFILEPICT );
    lpMFP = (LPMETAFILEPICT) GlobalLock( hGMem );
    SaveDC( hdc );
    CreateMapMode( hdc, lpMFP, cxWnd, cyWnd );
    PlayMetaFile( hdc, lpMFP->hMF );
    RestoreDC( hdc, - 1 );
    GlobalUnlock( hGMem );
    CloseClipboard();
}
```

The *ClipBoard* demo program includes several other examples where specific data types are requested. These are discussed further in the following sections.

Restrictions on Clipboard Operations

There are a few restrictions on clipboard operations:

- Before you can copy an item to the clipboard, you must call `EmptyClipboard` to erase the current contents of the clipboard. Remember, simply accessing the clipboard does not transfer ownership of the existing contents. Use the `EmptyClipboard` function to assign ownership and, at the same time, to clear (release) any and all existing contents.
- Any application can access the contents of the clipboard, but only the clipboard owner—an application that has called the `EmptyClipboard` function—can write material to the clipboard. However, because the clipboard can have only one owner, the previous owner's contents are simply erased, even if the same application was the previous owner.
- Although you can copy multiple items to the clipboard, you must transfer all of them in a single operation. The clipboard cannot be opened, written, closed, and then reopened again to transfer another item (at least not without erasing the first item transferred).
- Only one item of each type can be transferred to the clipboard at any time. This is for the simple reason that there is no method to distinguish between multiple items of a given type. However, when multiple items of different types have been written to the clipboard, an application accessing the clipboard may request only one item, several items, or all items, but it must request each item separately.

TIP

If an application desires to preserve the original contents of the clipboard while adding new material, the simple solution is to paste the existing contents, and then re-copy them to the clipboard together with whatever new material is desired.

The clipboard can be opened repeatedly to request different items or to request the same item a second (or third, fourth, and so on) time. But, in general, when requesting an item from the clipboard, the best option is to make a local copy of the desired item rather than attempting to request the same item more than once. Remember, there are no assurances that the data item requested will remain available locally.

The Office 2000 Clipboard

The Microsoft Office 2000 clipboard (the clipboard feature as of Office 2000, used with Microsoft Word, Excel, and so on) is no longer the same as the Windows clipboard. While the two function in a similar manner, and the contents of one clipboard can be accessed from the other, the Office 2000 clipboard has added capabilities for storing multiple items of the same type as well as for providing a Clipboard Toolbar for selecting items for paste.

Please note that these features are specific to Microsoft Office 2000 and are not inherent in Windows 2000, nor can they be readily included in your applications.

The *Clipboard* Demo: Reading and Writing Different Data Types

The *Clipboard* demo demonstrates writing to and reading from the clipboard with three different data types: text, bitmap, and metafile. *Clipboard* uses a simple menu with two primary options, Data To Clipboard and Data From Clipboard, each with a submenu listing equally straightforward Write and Retrieve options.

The Write Bitmap option includes a simple provision that captures the entire screen (or at least a 640×480 section of the screen) as a bitmap, writing the image to the clipboard. The Write Metafile option uses the same metafile construct demonstrated in the *PenDraw6* demo (described in Supplement 21). The Write Text option copies a simple text string to the clipboard.

NOTE

The *Clipboard* demo is included on the CD that accompanies this book, in the Supplement 22 folder.

Clipboard Text Transfers

Text operations may be the simplest type of clipboard operation, if only because text (string) operations themselves are comfortably familiar and require little explanation.

Writing Text to the Clipboard

Because the *Clipboard* demo will be both source and recipient, the first step is to transfer text information to the clipboard. The text chosen is a brief static string, declared as, “The quick brown fox jumps over the lazy red dog.” (It’s not very original, but it serves to demonstrate the principles involved.)

The mechanism for handling the text transfer is provided by a subprocedure called with two parameters: a handle to the application (hwnd) and a pointer to the text string (lpText).

```
BOOL TextToClipboard( HWND hwnd, LPSTR lpText )
{
    int          i, wLen;
    GLOBALHANDLE hGMem;
    LPSTR        lpGMem;
```

Within the `TextToClipboard` subroutine, four local variables are required, although only the latter two need an explanation. The `hGMem` variable provides a global handle to a memory block that has not yet been allocated. The second variable, `lpGMem`, is used as a pointer into the memory block.

After the `wLen` variable is initialized with the length of the text parameter, `hGMem` becomes a pointer to memory that is globally allocated to hold a copy of the text. Notice, however, that `wLen` is one character larger than the string, providing space allocation for a null terminator. Clipboard text is always stored in ASCIIZ (or ANSIZ) format:

```
wLen = strlen( lpText );
hGMem = GlobalAlloc( GHND, (DWORD) wLen + 1 );
lpGMem = GlobalLock( hGMem );
```

Last, `lpGMem` receives the pointer to the memory block returned by the `GlobalLock` function. But remember that the `GHND` specification has properly declared this memory block as movable. Also, in addition to returning an address (which could have been obtained several other ways), the `GlobalLock` function locks the memory block, temporarily preventing it from being moved by Windows.

The second feature provided by the GHND specification is to clear the memory block allocated, filling the memory block with zeros. Thus, the next step is to copy the local string, pointed to by `lpText`, into the memory block.

```
for( i=0; i<wLen; i++ )
    *lpGMem = *lpText++;
GlobalUnlock( hGMem );
return( TransferToClipboard( hwnd, hGMem, CF_TEXT ) );
}
```

After copying the text information, `GlobalUnlock` is called to release the lock on `hGMem`, making it movable and relocatable. However, if the memory block had been moved while the local text information was being copied, the `lpGMem` pointer would not have remained valid.

Memory ownership and the Clipboard

Do not under any circumstances free memory after transferring a data object to the clipboard. For example, imagine if the text-to-clipboard operation were rewritten to include a `GlobalFree` instruction, thus:

```
for( i=0; i<wLen; i++ )
    *lpGMem = *lpText++;
GlobalUnlock( hGMem );
GlobalFree( hGMem );
return( TransferToClipboard( hwnd, hGMem, CF_TEXT ) );
}
```

In this example, the result of calling `GlobalFree` would delete the item from the clipboard. Instead, once a data item has been transferred, ownership of the item has also been transferred and the local handle should not be used or tampered with further.

Likewise, when a data object is retrieved from the clipboard, the handle to the retrieved data may be locked and unlocked as necessary, but should not be freed because the object itself still belongs to the clipboard.

Data objects placed on the clipboard are freed only when an application assumes ownership of the clipboard and calls the `EmptyClipboard` instruction. At this point, all data objects owned by the clipboard are freed by the clipboard itself.

As a last step, the `TransferToClipboard` function (discussed earlier in this supplement) is called with the `hGMem` block, the flag `CF_TEXT`, and the application's window handle to complete the transfer process.

NOTE

In the *Clipboard* demo, only one item at a time is written to the clipboard. If you want to experiment, you can try adding a provision to copy multiple items.

Retrieving Text from the Clipboard

Retrieving text from the clipboard is almost as simple as writing it, but instead of a subroutine, the text-retrieval operations in the *Clipboard* demo are included in the response to the `WM_PAINT` message. This allows the application to update the window as required. (Other applications may handle this in another fashion.)

Retrieval operations begin by opening the clipboard and using the `GetClipboardData` API call to return a handle to the clipboard memory block.

```
OpenClipboard( hwnd );  
hTextMem = GetClipboardData( CF_TEXT );  
lpText = GlobalLock( hTextMem );
```

Just as was done during the transfer to the clipboard, and for the same reasons, `GlobalLock` is called to lock the memory block, returning a pointer to the memory address held by `lpText`. This time, however, instead of a loop, the `lstrcpy` function is used to copy the string contents from the memory address (`lpText`) to the local variable, `TextStr`.

```
lstrcpy( TextStr, lpText );  
GlobalUnlock( hTextMem );  
CloseClipboard();
```

Last, `GlobalUnlock` releases the lock on the memory block while `CloseClipboard` completes the operation. It's important to remember that a memory block should never be left locked; always call `GlobalUnlock` after calling `GlobalLock`.

Bitmap Clipboard Transfers

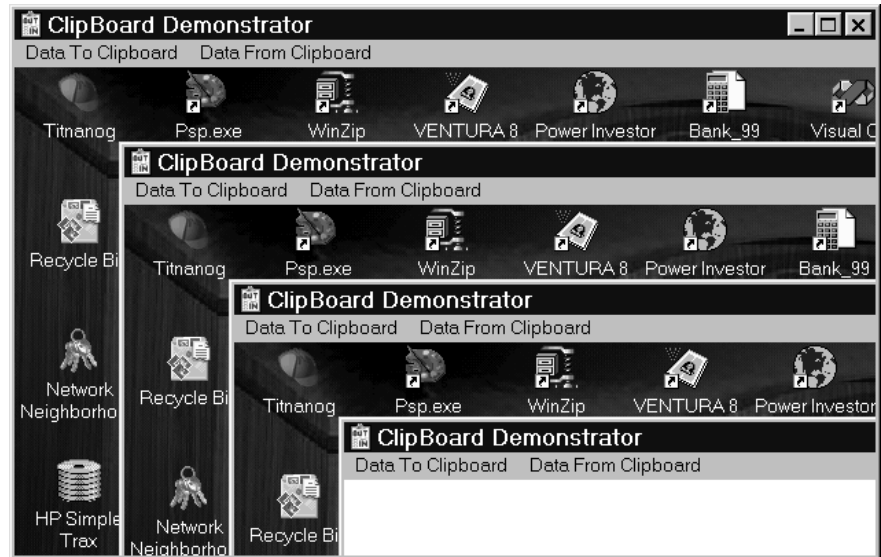
The *Clipboard* demo also includes a demonstration of bitmap transfer through the clipboard. This demonstration begins with provisions to capture the existing screen to provide a bitmap for transfer to the clipboard. The screen-capture

process itself is peripheral to the present topic; it follows the general form demonstrated in the *Capture* demo (discussed in Supplement 16).

Figure S22.1 shows the *Clipboard* demo window in a recursive situation following several screen captures.

FIGURE S22.1:

A recursive clipboard view



Writing a Bitmap to the Clipboard

The bitmap-to-clipboard transfer process begins by creating a compatible device context, `hdcMem`, in memory and then creating and selecting a compatible bitmap (also in memory).

```
hdc = GetDC( hwnd );
hdcMem = CreateCompatibleDC( hdc );
hBitmap = CreateCompatibleBitmap( hdc, 640, 480 );
SelectObject( hdcMem, hBitmap );
```

The next step is to copy the bitmap image from the original (the source)—in this example, the screen—to the memory context before calling the `TransferToClipboard` function to complete the transfer.

```
StretchBlt( hdcMem, 0, 0, 639, 479,
            hdc, 0, 0, 639, 479, SRCCOPY );
```

```
TransferToClipboard( hwnd, hBitmap, CF_BITMAP );  
DeleteDC( hdcMem );
```

Last, as cleanup, the memory device context is deleted, leaving ownership of the bitmap image to the clipboard.

Retrieving a Bitmap from the Clipboard

Retrieving the bitmap image from the clipboard is similar to the process of copying the image to the clipboard. The process begins by opening the clipboard and retrieving a handle to the bitmap (from the clipboard):

```
OpenClipboard( hwnd );  
hBitmap = GetClipboardData( CF_BITMAP );  
hdcMem = CreateCompatibleDC( hdc );  
SelectObject( hdcMem, hBitmap );
```

Again, a compatible device context is required. Then the `SelectObject` function selects the bitmap.

At this point, there are a few other tasks involved. First, the mapping mode needs to be set in the memory context for compatibility with the display context. And, second, before the bitmap can be copied, the size of the bitmap is needed. The bitmap size is obtained by copying the bitmap header into a local variable, `bm`.

```
SetMapMode( hdcMem, GetMapMode( hdc ) );  
GetObject( hBitmap, sizeof(BITMAP), (LPSTR) &bm );
```

The `BITMAP` variable (`bm`) now contains the bitmap header information needed to copy the actual bitmap from the memory context to the device context, this time using the `BitBlt` function.

```
BitBlt( hdc, 0, 0, bm.bmWidth, bm.bmHeight,  
        hdcMem, 0, 0, SRCCOPY );  
ReleaseDC( hwnd, hdc );  
DeleteDC( hdcMem );  
CloseClipboard();
```

All that's left is a bit of cleanup before closing the clipboard, and the job is done.

NOTE

The clipboard could have been closed earlier—just as soon as a handle had been returned to the image memory block—because closing the clipboard doesn't delete the memory block.

Metafile Clipboard Transfers

Metafile transfers introduce an element that is not present in text transfers. Although this element is present in bitmap transfers, it is not obtrusively visible, because it was handled virtually without remark. The new element required for metafile transfers is information about the file: the mapping mode under which the metafile was originally created and the extent or size information. This data is not necessarily inherent in the metafile itself.

With a text, metric, English, or TWIPS mapping mode, the mapping scale is fixed. The isotropic and anisotropic mapping modes, both of which have advantages for metafile operations, present a need for special information to accompany the metafile instructions.

For metafile clipboard transfers, the `METAFILEPICT` record is used. This record includes a record of the mapping mode used, size information, and the metafile script itself.

The `METAFILEPICT` record structure is defined in `WinGDI.H` as:

```
typedef struct tagMETAFILEPICT
{
    LONG        mm;
    LONG        xExt;
    LONG        yExt;
    HMETAFILE    hMF;
} METAFILEPICT, FAR *LPMETAFILEPICT;
```

The `mm` field contains the mapping mode. The `hMF` field is a handle to the metafile instructions. The remaining two fields, `xExt` and `yExt`, may contain two different types of information, depending on the mapping mode.

For text, metric, English, or TWIPS mapping modes, the `xExt` and `yExt` fields specify the horizontal and vertical size of the metafile picture in units appropriate to the mapping mode.

For the `MM_ISOTROPIC` or `MM_ANISOTROPIC` mapping modes, the `xExt` and `yExt` fields contain an optional suggested size expressed in `MM_HIMETRIC` units, or these fields may be zero if no suggested size is offered. Alternatively, if the `xExt` and `yExt` fields are negative, the information is provided as a suggested size ratio, rather than an absolute size.

In the *Clipboard* demo, a metafile image is supplied by duplicating the metafile image code from Supplement 21, now in a subroutine titled `DrawMetafile`. Much

of the `DrawMetafile` procedure should be familiar from previous examples, but it does begin with one new variable declaration:

```
BOOL DrawMetafile( HWND hwnd, int cxWnd, int cyWnd )
{
    LPMETAFILEPICT lpMFP;
```

Aside from the variable `lpMFP`, which is used as a pointer to an instance of the `METAFILEPICT` structure, the `DrawMetafile` function proceeds by creating a metafile in memory as demonstrated in Supplement 21. But after the metafile is created, the next step is to create a `METAFILEPICT` structure in memory and, using `GlobalLock`, to return a value to the `lpMFP` pointer.

```
hGMem = GlobalAlloc( GHND, (DWORD) sizeof( METAFILEPICT ) );
lpMFP = (LPMETAFILEPICT) GlobalLock( hGMem );
```

Now that the `METAFILEPICT` structure is allocated and locked, the next steps are to assign values to the mapping mode, provide a suggested size, and assign the metafile handle.

```
lpMFP->mm = MM_ISOTROPIC;
lpMFP->xExt = 200;           // suggested size in //
lpMFP->yExt = 200;           // MM_HIMETRIC units //
lpMFP->hMF = hMetaFile;
```

And, with the assignments completed, all that remains is to unlock the memory before transferring the handle to the clipboard.

```
GlobalUnlock( hGMem );
TransferToClipboard( hwnd, hGMem, CF_METAFILEPICT );
```

Retrieving a Metafile from the Clipboard

Retrieving the metafile from the clipboard begins by opening the clipboard, asking for a handle to the memory block containing the metafile, and locking the block while returning a pointer.

```
OpenClipboard( hwnd );
hGMem = GetClipboardData( CF_METAFILEPICT );
lpMFP = (LPMETAFILEPICT) GlobalLock( hGMem );
```

At this point, the `lpMFP` variable contains a pointer to the metafile memory block—or, more accurately, to the `METAFILEPICT` structure—which contains a pointer to the metafile proper.

But before replaying the metafile itself, there are a couple of other tasks that require attention, beginning by saving the present device context (as suggested in Supplement 21).

```
SaveDC( hdc );
CreateMapMode( hdc, lpMFP, cxWnd, cyWnd );
```

After saving the present device context, the `METAFILEPICT` information is passed to a subroutine, `CreateMapMode`, for processing. This is done because deciphering the mapping mode and size/extent information is moderately complex.

`CreateMapMode` is called with four parameters: the application's device-context handle, the `METAFILEPICT` pointer, and the application window's size.

```
BOOL CreateMapMode( HDC hdc,      LPMETAFILEPICT lpMFP,
                    int cxWnd, int cyWnd )
{
    long lMapScale;
    int  nHRes, nVRes, nHSize, nVSize;

    SetMapMode( hdc, lpMFP->mm );
    if( lpMFP->mm != MM_ISOTROPIC && lpMFP->mm != MM_ANISOTROPIC )
        return( TRUE );
}
```

First, `CreateMapMode` sets the mapping mode specified for the metafile. If the mapping mode is anything except `MM_ISOTROPIC` or `MM_ANISOTROPIC`, the function simply returns, with nothing more required.

NOTE

The image size data could be extracted, but there really isn't any point in doing so in this demo. If you wish, you could get the size information and use it to position the metafile image. (See Supplement 21.)

If the metafile-mapping mode is `MM_ISOTROPIC` or `MM_ANISOTROPIC`, then the `CreateMapMode` function still has work to do. First, it proceeds by calling the `GetDeviceCaps` function to query the horizontal and vertical size and resolution.

```
nHRes = GetDeviceCaps( hdc, HORZRES );
nVRes = GetDeviceCaps( hdc, VERTRES );
nHSize = GetDeviceCaps( hdc, HORZSIZE );
nVSize = GetDeviceCaps( hdc, VERTSIZE );
```

The next actions depend on the values passed in the `xExt` and `yExt` fields. These values may be positive or negative, or no values at all may be passed. If the

arguments are positive, the values are intended to suggest a size in MM_HIMETRIC units. Therefore, the `SetViewportExtEx` function is called to set the viewport size appropriately.

```
if( lpMFP->xExt > 0 )
    SetViewportExtEx( hdc,
        (int)((long) lpMFP->xExt * nHRes / nHSize / 100 ),
        (int)((long) lpMFP->yExt * nHRes / nHSize / 100 ),
        NULL );
```

If negative values have been entered, the arguments are intended as a ratio rather than an absolute size. Therefore, the first step is to calculate a scale to fit the device context.

```
else
    if( lpMFP->xExt < 0 )
    {
        lMapScale = min( ( 100L * (long) cxWnd * nHSize / nHRes
                        / -lpMFP->xExt ),
                        ( 100L * (long) cyWnd * nVSize / nVRes
                        / -lpMFP->yExt ) );
```

Two scales are calculated: one to fit the x-axis and one to fit the y-axis. But the `iMapScale` value is chosen as the smaller of the two possible scales to ensure that the resulting image fits the display. Once the mapping scale has been calculated, `SetViewportExtEx` is called again to size the viewport to fit.

```
        SetViewportExtEx( hdc,
            (int)((long) -lpMFP->xExt * lMapScale * nHRes
                / nHSize / 100 ),
            (int)((long) -lpMFP->yExt * lMapScale * nVRes
                / nVSize / 100 ), NULL );
    }
```

The third possibility is that neither size nor ratio information was supplied. In this case, the solution is simply to set the viewport to match the window size.

```
else
    SetViewportExtEx( hdc, cxWnd, cyWnd, NULL );
```

Once the `CreateMapMode` function returns, the remainder of the task is simple, requiring nothing more than a call to `PlayMetaFile`, almost exactly as shown previously.

```
PlayMetaFile( hdc, lpMFP->hMF );
RestoreDC( hdc, - 1 );
```

```
GlobalUnlock( hGMem );  
CloseClipboard();
```

And, after replaying the metafile, the `RestoreDC` function is called with an argument of `-1` to restore the original device context, the metafile memory is unlocked, and the clipboard is closed.

Using Other Clipboard Formats

The three clipboard formats used in the *Clipboard* demo demonstrate the general processes involved in working with the clipboard. Any of the other clipboard formats should present no special difficulties. However, there are a few clipboard formats that merit some explanation, not because they require special handling, but because they are defined for special purposes.

Private Clipboard Formats

Windows defines the “private” clipboard formats of `CF_DSPTEXT`, `CF_DSPBITMAP`, `CF_DSPMETAFILEPICT`, and `CF_DSPENHMETAFILE`. These correspond to the `CF_TEXT`, `CF_BITMAP`, `CF_METAFILEPICT`, and `CF_ENHMETAFILE` types, but with one principal difference: Applications requesting standard formats will not access these private formats. There are two assumptions here:

- Data using one of these private formats is intended for exchange between two instances of the same application or two applications specifically designed to operate together.
- Such exchanges may include private information, such as formatting and/or font information used by the Windows Write program.

The term *private* could be misleading. There is nothing to prevent any application from requesting access to one or more of these private formats; these are not designed for security purposes, only to declare nonpublic clipboard transfers.

Also, although two instances of an application (or two related applications) should understand their own private formats, the use of one of these formats does not ensure that the originator is indeed another instance of the same application or a companion application. In other words, there is nothing to prevent another totally unrelated application from using these same format designations.

However, provisions have been made for this circumstance and you can obtain the originator of the clipboard contents by calling the `GetClipboardOwner` function:

```
hwndCBOwner = GetClipboardOwner();
```

When the `EmptyClipboard` function was called in preparation for copying materials to the clipboard, the calling application became the new clipboard owner. Other applications accessing the clipboard to retrieve material do not gain ownership of the clipboard, only access. Thus, only the clipboard owner is responsible for originating material on the clipboard, and an application cannot place material on the clipboard without first becoming the clipboard owner and erasing the previous contents. Therefore, any application accessing information in a private format should also query the identity of the clipboard owner to determine if this data is indeed in a common format.

Still, although the `GetClipboardOwner` function returns a handle identifying the owner, this handle doesn't really tell you very much. But given the handle, you can make another call to query the application's class name:

```
GetClassName( hwndCBOwner, &szClassName, 16 );
```

Finally, you can compare `szClassName` with the current application's class name or to a list of companion application class names to identify the source of the clipboard information.

Delayed Rendering

Posting data to the clipboard frequently involves passing a copy of the data to the clipboard while keeping the original intact, which means expending memory on duplicate data blocks. In many circumstances, this may be unimportant, particularly when the memory requirements are small. One obvious solution, which is used in the *Clipboard* demo, is to transfer the data without keeping a copy, thus avoiding the problem entirely.

There is also another solution, which is particularly appropriate when large amounts of data are involved: delayed rendering of the clipboard data. In delayed rendering, only the format specification is posted to the clipboard and, instead of a global memory block handle, the handle parameter is passed as a `NULL`:

```
SetClipboardData( wFormat, NULL );
```

When an application requests a data item that has been posted for delayed rendering, identified by the `NULL` in place of the data block, Windows recognizes the

use of delayed rendering. Windows then calls the clipboard owner (the application that posted the material to the clipboard) with a `WM_RENDERFORMAT` message, with the requested format specified in `wParam`.

In response to the `WM_RENDERFORMAT` message, the application is expected to respond with a `SetClipboardData` call, accompanied by the global memory block handle and the format identifier (rather than responding with an `OpenClipboard` and `EmptyClipboard` call). In this fashion, the actual data is posted only when the recipient is ready to accept it.

You may pass multiple items to the clipboard as a mixture of conventional data transfers and delayed rendering transfers.

Special Circumstance Messages

When an application loses ownership of the clipboard, Windows does nothing to prevent this loss but does post a `WM_DESTROYCLIPBOARD` message to the previous owner, indicating that ownership has been lost. In response, an application can resume ownership and post the same material again, but this is not recommended except in special circumstances.

Also, if an application is ready to terminate itself but is also currently the clipboard owner and the clipboard contains `NULL` data handles, Windows will send a `WM_RENDERALLFORMATS` message, without any format specifications, before the application is permitted to terminate. In response, the owner application has two options: clear the clipboard entirely or complete the delayed calls.

Unlike the response to the `WM_RENDERFORMAT` call, however, the terminating application should not use the `SetClipboardData` call but should simply clear the clipboard and write new entries entirely, just as if delayed rendering had not been used at all.

Owner-Displayed Clipboard Data

Another, very private, clipboard format is declared as:

```
SetClipboardData( CF_OWNERDISPLAY, NULL );
```

The `CL_OWNERDISPLAY` type is always passed with the global memory handle specified as `NULL`, just as with the delayed rendering format. But, because the clipboard owner is directly responsible for the display, Windows does not send a `WM_RENDERFORMAT` message when the data is requested. Instead, messages must

be sent directly from the clipboard viewer to the clipboard owner. See the discussion of the other private clipboard formats for ways to identify the clipboard owner using the `GetClipboardOwner` function. Conversely, the clipboard owner can use the `GetClipboardViewer` function, if necessary, to identify the viewing application.

To use this private format, the viewer application would post a request to the clipboard owner application, requesting the originating application to provide the actual display and granting the originating application access to the destination application's display. Five messages may be sent from the destination:

WM_ASKCBFORMATNAME This message is sent by the clipboard viewer to request a copy of the format name from the clipboard owner. Remember, the clipboard itself contains only the `CF_OWNERDISPLAY` identifier, and the viewer application is still free to decide if it is interested in the actual data type. The `WM_ASKCBFORMATNAME` message is accompanied by a specification in `wParam` for the number of bytes to copy. `lParam` provides a pointer to the buffer where the response should be posted.

WM_HSCROLLCLIPBOARD/WM_VSCROLLCLIPBOARD These messages are sent when the viewer application contains a horizontal or vertical scrollbar and a scrollbar event must be reported to the clipboard owner. The `wParam` argument contains a handle to the viewer's window. The `lParam` argument contains the same scrollbar messages that would accompany standard `WM_HSCROLL` or `WM_VSCROLL` messages.

WM_PAINTCLIPBOARD This message is sent requesting a repaint of the viewer application's display, probably in response to a `WM_PAINT` message received by the viewer application. The `wParam` argument contains a handle to the viewer's window. The `lParam` argument is a global `DDESHARE` handle that, when locked, points to a `PAINTSTRUCT` structure defining the area requiring repainting. To determine if all or part of the client area requires repainting, the clipboard owner must compare the dimensions of the drawing area reported in the `rcpaint` field of the `PAINTSTRUCT` with the dimensions reported in the most recent `WM_SIZECLIPBOARD` message.

WM_SIZECLIPBOARD This message is sent to indicate that the clipboard viewer has changed size. The `wParam` argument contains a handle to the viewer window. The `lParam` argument is a global `DDESHARE` handle pointing to a `RECT` structure defining the area to be painted.

In response to any of these messages, the clipboard owner should use the `InvalidateRect` function or repaint the viewer as desired, and reset the scrollbar positions appropriately.

User-Defined Private Formats

Applications may also define their own private clipboard formats, registering a new clipboard format by calling the `RegisterClipboardFormat` function:

```
wFormat = RegisterClipboardFormat( lpszFormatTitle );
```

The returned `wFormat` identifier will be a value in the range `0xC000` to `0xFFFF` and can subsequently be used as the format parameter in `SetClipboardData` and `GetClipboardData` calls. Before another application or instance can retrieve clipboard data in this format, it will require the same `wFormat` ID. This value could be passed via the clipboard using the `CD_TEXT` format.

Alternatively, you could use the `EnumClipboardFormats` function, discussed earlier in the supplement, to return all format identifiers, after which you should call the `GetClipboardFormatName` function to return the ASCII name of the format:

```
GetClipboardFormatName( wFormat, lpszBuffer, nCharCount );
```

WARNING

The format identifiers `CF_PRIVATEFIRST` (`0x0200`) and `CF_PRIVATELAST` (`0x02FF`) may also be used as a range of integer values for private format identifiers. Note that data handles associated with formats in this range will not be freed automatically when another application requests clipboard ownership. Instead, any data handles in this range must be freed by the owner application before the application terminates or when a `WM_DESTROYCLIPBOARD` message is received. Use these latter format IDs with care.

Finally, note that Windows does not require any information about the organization of the data transferred using a private format. It is the application's responsibility to understand the details of the transfer format. All that Windows requires is a format name and a handle to the memory block.

Summary

As you've seen in this supplement, a variety of standard and special-purpose clipboard formats are available, or applications can define and register their own special formats. The complete listing for the *Clipboard* program, which demonstrates working with text, bitmap, and metafile formats, is on the CD that accompanies this book.

S U P P L E M E N T

T W E N T Y - T H R E E

S23

OLE Client and Server Application Development

- OLE basics
- OLE library functions
- OLE server registration and selection
- OLE client development
- OLE server application development

You've learned about the clipboard and DDE in the previous supplements. *Object Linking and Embedding (OLE)* provides yet another way for applications to share data. OLE has the advantage of being virtually unlimited in its scope. An OLE application you write now will work perfectly well even if it encounters a server that supplies data in a format Microsoft hasn't anticipated. Microsoft doesn't need to anticipate formats; if a server can handle the data, any client can receive it. A user may well apply OLE programs to tasks the developer never imagined.

Introducing OLE

OLE is a set of protocols and procedures proposed by Aldus Corporation in 1988 to simplify the creation and maintenance of compound documents. A *compound document* is a file belonging to one application (for example, a word processor) that also includes data created by another application (such as a graphics editor). Blocks of foreign data in a compound document are called *objects*. An application that receives data objects and builds compound documents is called an OLE *client*, and one that exports objects for other applications to use is called an OLE *server*. Whether an application is a client or a server depends on its role in a particular interaction. One application may act simultaneously as a client and a server in different interactions.

Application-Based versus Document-Based Environments

When Microsoft built OLE into Windows, it took a big step toward making the user's work center on documents rather than applications. Traditionally, the user invokes a single application for each new document. Changing from one data format to another—from text to numbers, or from pictures to sounds—usually means quitting one application and starting another. Typically, in an application-based environment, a document makes sense only when read by the application that created it.

A document-based environment, on the other hand, lets several applications cooperate in creating a single document. No one application understands all the

objects in the document, but as you move from piece to piece, the system automatically invokes the appropriate applications. You edit the pieces separately in their native applications, and the master document automatically receives updates from every contributor. You have more freedom to exercise creativity in combining sounds, video, pictures, numbers, and text in a single, integrated document. You can show pictures in your word processor or attach video clips to records in your database.

Compound documents existed in Windows before OLE, but their capabilities were limited. A user would create a compound document by copying data to the clipboard and pasting it into another application. In this common transaction, a data object moves from a server to a client program. But whenever the server subsequently edits its copy of the object, the cut-and-paste operation must be repeated *for all documents into which the object has been pasted*.

To create a document-based environment, the system must offer substantial facilities for coordinating applications. For example, the system must know which applications can operate on which kinds of data. As the user moves from object to object through a document, the system must recognize and support links to various other programs. In Windows, the OLE extension libraries assume these complex chores. The three libraries that implement OLE currently contain a variety of functions to help you create programs that handle virtually any kind of data object through a seamless cooperation with the program that created it.

Linking versus Embedding

An object is any set of data from one application treated as a unit. OLE applications create compound documents when they combine several objects into one file. The user sees all the objects from one document displayed together in a single window. To the client program, each object looks like a black box full of incomprehensible data. The program calls OLE functions to manipulate objects; it does not need to understand them.

When importing an object, a client chooses between linking and embedding. *Embedding*, just like pasting, gives the client a complete and independent copy of the data. An embedded object, however, remembers its origin, and the user can edit an embedded document by double-clicking it. The double-click invokes the server application, and the editing happens there. When the user closes the server, the client receives the updated object.

The second way of storing objects is to *link* them. Linking does not give the client its own independent copy of the data; instead, the client receives a live connection to a piece of the server's document—a kind of window opening into a view of the server's data. If the object is modified in the server, the modifications appear automatically in the client. If several clients link to the same object, an update in one place is visible in all the others.

In summary, when a document file contains all the data for an object, the object is *embedded*. When a document contains only a reference pointing to data in another document, the object is *linked*. Both methods produce the same result on the screen, but only linked objects receive updates. Embedded objects are transferred through the equivalent of a DDE cold link; when you copy them into your document, they become independent of their source. By contrast, if you link an object into several documents, changing the object in one place causes it to change in all the others.

Linked objects have the advantage of taking less space in the document file. Embedded objects have the advantage of being very portable; documents that contain only embedded objects can move from system to system. Because they carry all their data with them, they do not require the OLE server to reside on the destination system. The user can change linked objects into embedded objects at will.

OLE Clients versus OLE Servers

As mentioned earlier, OLE applications come in two basic types: clients and servers. If you embed a Paintbrush picture in a Write document, Paintbrush is the server and Write is the client. Write does not need to understand the data that makes up the image file. Write calls OLE functions to display the data. If the OLE system doesn't know how, it calls on the server, Paintbrush, to display data in the Write window.

Applications such as Word, WordPad, Excel, and Quattro Pro are both clients and servers. These applications both accept OLE objects supplied by other servers and act as OLE servers to other client applications. The Write editor and CardFile programs under Windows 3.x function only as OLE clients; they do not offer server services. (Few contemporary applications act as clients without also offering server capabilities.) In contrast, the Windows Paint program is an OLE server but incorporates no client capabilities. Instead, the Paint program limits itself to providing images and image-editing services to client applications.

Because Paint is a stand-alone application as well as an OLE server, Paint is a *full server*. In contrast, a *mini-server* does not operate as a stand-alone application; it does not contain any provisions to open or save files, but it does provide services to OLE client applications.

Servers of either type (full server or mini-server) may offer more than one type of service. Quattro Pro, for example, offers a choice of Quattro Pro Graph or Quattro Pro Notebook. Microsoft Word offers a choice of Microsoft Word Document or Microsoft Word Picture objects.

Object Classes and Verbs

The type of data a server exports is called an *object class*. Different classes contain different kinds of data. Paintbrush, for example, exports objects of the PBrush class. Excel supports the classes ExcelWorksheet and ExcelChart. Servers register their classes in the system registry. Only one server may handle each class.

For each of its object classes, a server also registers a set of verbs. A *verb* is something a server can do to an object. Two common verbs are Edit and Play. When the user selects an object in a compound document, the client application retrieves the list of verbs for that object class and makes the verbs available on one of its menus. The user manipulates objects by executing their verbs. Different objects respond to different verbs.

Inserting OLE Objects

The process of adding an object to a container document is simple. From within the client application, the user chooses the type of object to insert. For example, a list box might offer picture data, spreadsheet data, and video clips; the list varies with the available servers.

Suppose that you're running Microsoft Word and decide to embed a drawing in your document. You start Paint, the registered server for bitmap objects. Then you open a .BMP file, select part of the image, and copy it to the clipboard. In Microsoft Word, you open the destination file and pull down the Edit menu. There you see a choice of three commands: Paste, Paste Link, and Paste Special. All of them bring the drawing into the text file. The easiest one, Paste, embeds the object. (If Paint did not support OLE, the Paste command would merely copy the object, not embed it.) The Paste Link option, of course, creates a link to the object source.

NOTE

The Paste Special command is part of the new Office clipboard and offers options for the format used to paste an object.

Inserting Object Packages

You can also embed .AVI or .WAV files (multimedia video or sound files) in a compound document. But what does a video clip or sound look like on the screen when you paste it? In this case, the application uses a graphical representation of the data called a *package*.

A package is an icon that represents an OLE object. When you double-click on the icon, the OLE libraries determine what data the object contains and perform the appropriate verb action.

For some data types, such as .WAV or .AVI files, only packages make sense. By default, the package icon comes from the program that created the data.

TIP

Using the Object Packager program that comes with Windows, you can customize both the icon and the label of any package.

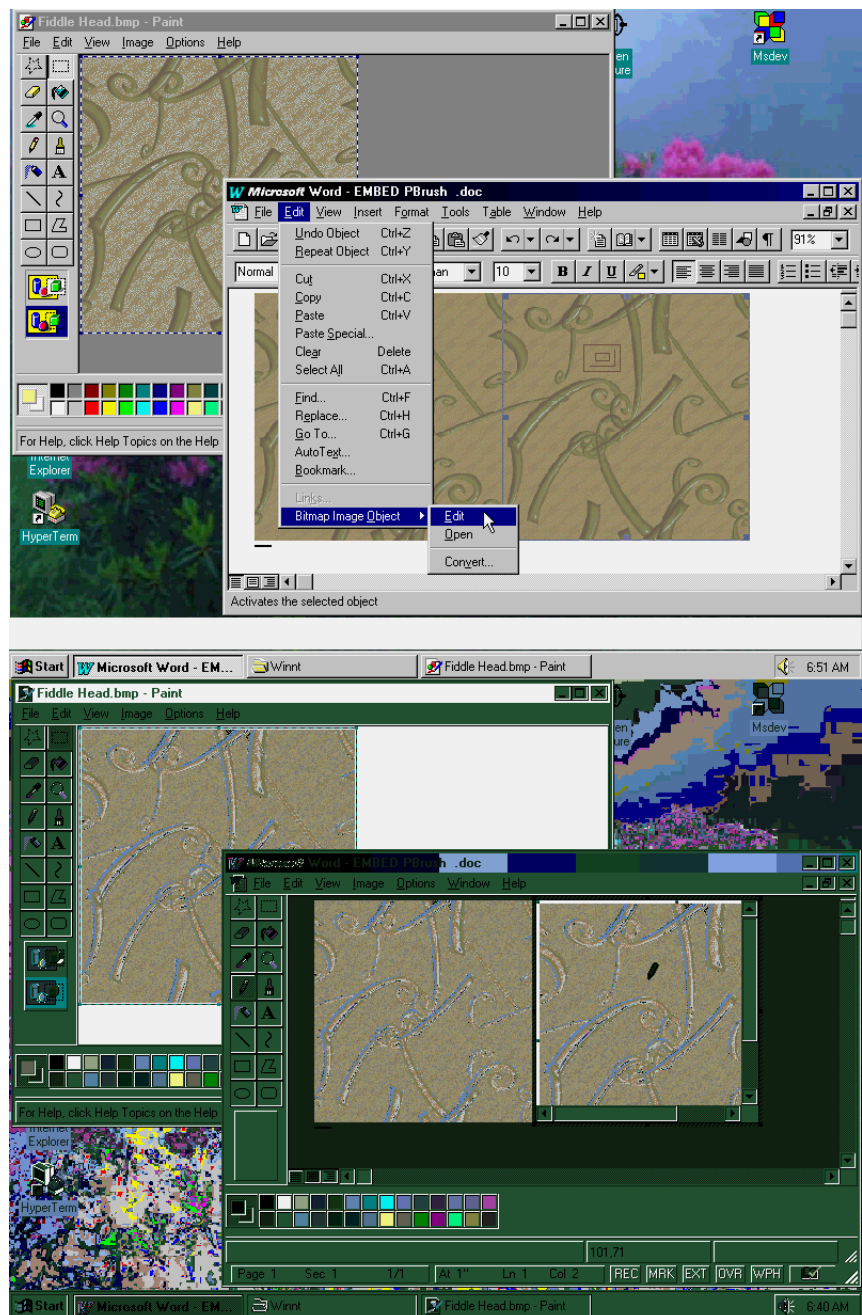
Working with Inserted Objects

Once an object is in the client's document, the client provides ways to activate it. Usually, double-clicking activates an object. An activated object performs whatever action is appropriate to its format.

Figure S23.1 shows the process of linking a picture into a Microsoft Word document, where OLE provides in-place editing (editing the picture directly within the linked document) using the Paint program.

FIGURE S23.1:

Linking a picture into a text document and activating the picture to edit it.

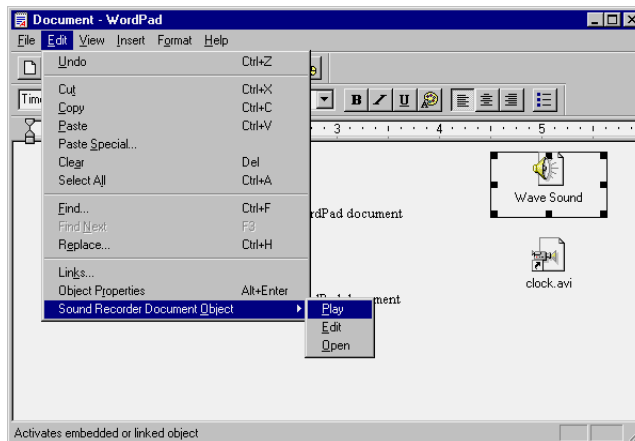


As illustrated, the Edit menu now contains a Bitmap Image Object submenu with three verbs—Edit, Open, and Convert—supplied by the Paint application. For a Paint drawing, the default is the verb Edit, which activates in-place editing. When you select in-place editing, the Paint menu, palette, and toolbars appear inside the Word document frame, allowing editing without leaving the document. Selecting Open calls the Paint application with the embedded image, as shown in the lower half of Figure S23.1. Changes made using the Paint program will be reflected in the embedded object in the Word document when the Paint program exits, signaling on termination that the compound document should be updated.

Figure S23.2 shows a sound package and a video package pasted into a WordPad document. The Edit menu, under the Sound Recorder Document Object entry, offers three verbs associated with sound data: Play, Edit, and Open. Since more than one OLE object is embedded in this document, the Wave Sound object must be selected before the Object Properties and Sound Recorder menu options are enabled.

FIGURE S23.2:

A WordPad document with embedded audio and video clips



The Clock.AVI file is also an embedded object and, when selected, provides the same three verb entries in the Edit menu under the Linked Video Clip Object entry. Both the .AVI and .WAV files also respond to a double-click: The .AVI object plays the video in a separate .AVI window, and the .WAV object plays the sound waveform through the system sound card (assuming that a sound card is installed).

Eventually, with the help of some OLE functions, the client application saves the compound document. The document can then be transferred from user to user and read by the same client application on other computers. If the new system lacks some servers, all the objects will still display correctly. This is because the OLE system itself handles standard clipboard formats like bitmaps and metafiles in any client without calling a server. You cannot activate objects without a server, however. You cannot do much with .WAV packages, for example, unless you have the Sound Recorder installed (and, of course, a sound card and speakers).

Storing Objects in Presentation and Native Data Formats

Two of OLE's goals seem to place contradictory demands on data objects. In order to display the object in any application, whether or not the original server is present in the system, the object must contain data in some common, recognizable display format, such as a metafile or bitmap.

On the other hand, OLE also lets the user continue to edit objects even *after* they are pasted into a new application. In order for the server to edit objects, they must contain whatever data the server uses to represent them internally. Excel, for example, cannot continue to edit spreadsheet cells that have been converted for display as a metafile, but neither can client programs—or even the OLE libraries—be expected to understand Excel's internal data well enough to display the cells by themselves on systems where Excel is not installed.

The solution is to supply two copies of the data for every OLE object. You'll read more in a moment about how the server does this, but essentially every OLE object contains data in a *native* format, as the server created it, and in one of several standard *presentation* formats—usually a metafile—so anyone can display it.

The OLE Libraries

OLE gives you high-level functions to implement low-level data-sharing processes, similar to the DDE functions provided by the DDEML. The OLE functions reside in three dynamic link libraries. OleCli32.DLL contains all the functions for an OLE client, and OleCvr32.DLL contains the functions for an OLE server. In

OLE 1.0, these two libraries exchange data and commands through DDE messages. The third library, Shell32.DLL, maintains a database of servers and data types to ensure that requests for assistance are routed correctly.

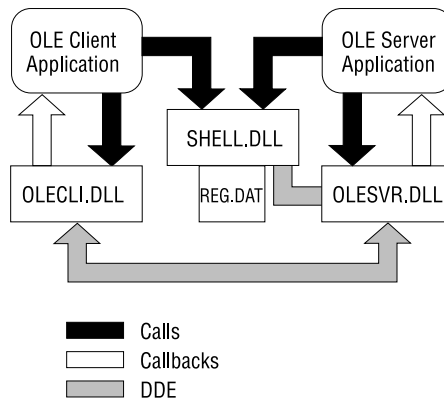
Interacting through the OLE Libraries

OLE applications interact with each other through the libraries. When a client decides to edit a picture object, for example, it passes the request to the OleCli DLL. OleCli sends a message to OleSvr. OleSvr locates a server and asks it to begin an editing session. When the user activates an object, the OleSvr library must determine which server application corresponds to the given data format. To identify servers, OleSvr consults the Shell library. The Shell functions manage the system registry, which maps each data type to a server application name. Servers add their own names to the registry during their installation.

When the operation has completed, OleSvr passes the results back to OleCli, and OleCli passes them on to the client. The interaction of client and server through the OLE libraries is shown in Figure S23.3.

FIGURE S23.3:

How the three OLE libraries interact with client and server applications



Choosing between the DDE and OLE Libraries

Like the DDEML, the OLE libraries work through the DDE protocol. OLE commands send DDE messages. The underlying DDE processes are invisible to an

OLE application. Because Microsoft developed the DDEML and OLE systems in parallel, neither relies on the other.

NOTE

Due to shrinking support for DDE and increasing emphasis on OLE under Windows 2000, DDE and DDEML are not discussed in this book.

To choose between the DDEML and OLE, consider what your application needs to do. For maintaining many links and updating them all frequently, choose the DDEML. One DDEML conversation can establish many links, but each OLE conversation transfers only a single object. Although OLE clients can initiate several conversations with one server, this incurs an overhead that the DDEML avoids. DDEML links, however, die when either participant terminates.

Choose OLE when you want to support any of the following:

- Persistent embedding and linking
- Rendering common data formats
- Rendering specialized data formats through the server
- Transferring data through the clipboard and through files
- Activating objects

Accessing OLE Information

As mentioned earlier, when an OLE server is installed on your system or is first executed, it registers itself with the Windows registry. This registration includes, among other elements, the name and location of the server, as well as the various types of services that it is prepared to supply.

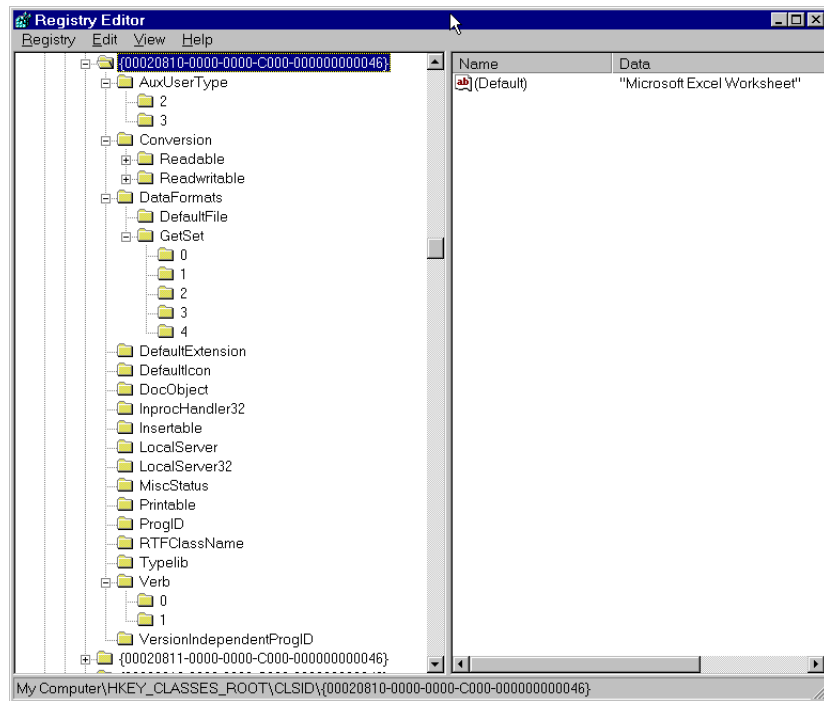
In turn, an OLE client application can query the registry to find an appropriate server and service. You can also access the registry information directly by using the Registry Editor utility (`RegEdit.EXE`, located in your `\windows` directory or `RegEdt32.EXE`, in your `\WINNT\SYSTEM` directory), as explained in Chapter 8, “Using the Registry.”

As an example, Figure S23.4 shows the Registry Editor after using the Edit > Find function to locate the Excel application registry information. Actually, there

are quite a few entries for Excel, but the one of interest is the class ID. This is found under the branch HKEY_CLASSES_ROOT\CLSID\ and identified by a unique (generated) class ID entry: 00020810-0000-0000-C000-000000000046. (As with most registry entries, this same information can also be found under the entry HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID, as well as in other locations; this duplication is not unusual.)

FIGURE S23.4:

The Registry Editor with information about Excel



As you can see in Figure S23.4, the available information includes data formats, conversion options, the default extension(s) and icon, the program ID, and, not least, which action verbs the OLE server supports.

TIP

Visual Studio includes an application titled OLEView (found in the Microsoft Visual Studio\Common\Tools directory), which provides another way to view information about OLE applications. Despite the lack of documentation, the OLE2View application does provide an interesting view of a variety of OLE-support functions.

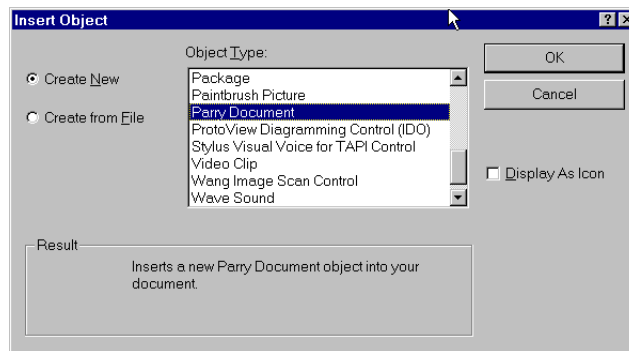
Most of the time, there is no need to access the registry directly. A number of mechanisms provide indirect access, such as the `COleInsertDialog` method (discussed later in this supplement), for safe (but restricted) access to registry information.

Selecting an OLE Server

Before an OLE client can use server services, the client application must select an OLE server. How a server is selected varies depending on how the client application chooses to set up the menu options. For the *Ole_Client* demo discussed in this supplement, the Edit menu's Insert New Object option calls the Insert Object dialog box, shown in Figure S23.5.

FIGURE S23.5:

The Insert Object dialog box

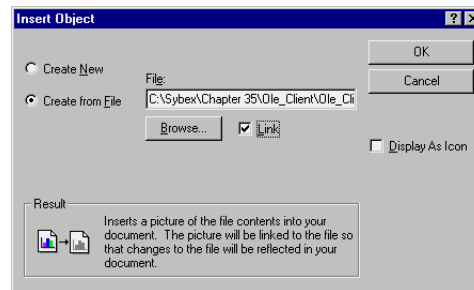


The list box in the Insert Object dialog box lists all of the registered OLE server object types. The Result box at the bottom offers a brief explanation of the selected item.

The Create New radio button is selected by default. If you select the Create from File radio button, the Insert Object dialog box changes to allow you to select a file of any type to insert as an OLE object, as shown in Figure S23.6. (Inserting a file does not guarantee that there is a supporting server for that file.) Here, you may enter path and filename information directly or click the Browse button to call the standard file-selection dialog box. If you select the Link checkbox, any changes to an OLE file object through external sources, such as when you edit the file through another application, are immediately reflected in the linked object.

FIGURE S23.6:

Inserting a file object



You can select the Display As Icon checkbox on the right side of the Insert Object dialog box to insert the object as an icon rather than as an active item. The advantage is that an “iconized” object does not require redrawing and remains inactive until selected. (Files are always inserted with icon representation, regardless of the selection made in the Display As Icon checkbox.)

NOTE

AppWizard supports full in-place editing for only OLE server objects, not for files. To fully support embedded or linked files, either the client application must be modified to provide support or the Edit menu’s Packager option must be used to call the appropriate support utility.

Registering an OLE Server

You’ve been introduced to the Registry Editor utility for viewing the registry and to the Insert Object dialog box for selecting a registered server. But how do you register an OLE server?

Full-server applications register themselves automatically the first time the server application is executed as a stand-alone process by invoking the `COleServerRegister` member. Furthermore, if the application was created using MFC and the AppWizard, the `COleServerRegister` function was installed in the `InitInstance` procedure as a call to the `COleTemplateServer::RegisterAll` function. These processes are described later in the supplement.

For mini-server applications, which cannot run as stand-alone applications, a different approach is necessary. AppWizard provides for registering a mini-server by creating a REG script for the server application:

```
REGEDIT
; This .REG file may be used by your SETUP program.
; If a SETUP program is not available, the entries below will be
; registered in your InitInstance automatically with a call to
; CWinApp::RegisterShellFileTypes and COleObjectFactory::UpdateRegistryAll.

HKEY_CLASSES_ROOT\Parry.Document = Parry Document
HKEY_CLASSES_ROOT\Parry.Document\protocol\StdFileEditing\server =
    PARRY.EXE
HKEY_CLASSES_ROOT\Parry.Document\protocol\StdFileEditing\verb\0 =
    &Edit
HKEY_CLASSES_ROOT\Parry.Document\Insertable =
HKEY_CLASSES_ROOT\Parry.Document\CLSID = {C6A0FC60-3173-11D0-93D7-
    BA6083000000}
HKEY_CLASSES_ROOT\CLSID\{C6A0FC60-3173-11D0-93D7-BA6083000000} =
    Parry Document
HKEY_CLASSES_ROOT\CLSID\{C6A0FC60-3173-11D0-93D7-
    BA6083000000}\DefaultIcon = PARRY.EXE,1
HKEY_CLASSES_ROOT\CLSID\{C6A0FC60-3173-11D0-93D7-
    BA6083000000}\LocalServer32 = PARRY.EXE
HKEY_CLASSES_ROOT\CLSID\{C6A0FC60-3173-11D0-93D7-BA6083000000}\ProgId
    = Parry.Document
HKEY_CLASSES_ROOT\CLSID\{C6A0FC60-3173-11D0-93D7-
    BA6083000000}\MiscStatus = 32
HKEY_CLASSES_ROOT\CLSID\{C6A0FC60-3173-11D0-93D7-
    BA6083000000}\AuxUserType\3 = Parry
HKEY_CLASSES_ROOT\CLSID\{C6A0FC60-3173-11D0-93D7-
    BA6083000000}\AuxUserType\2 = Parry
HKEY_CLASSES_ROOT\CLSID\{C6A0FC60-3173-11D0-93D7-
    BA6083000000}\Insertable =
HKEY_CLASSES_ROOT\CLSID\{C6A0FC60-3173-11D0-93D7-BA6083000000}\verb\1
    = &Open,0,2
HKEY_CLASSES_ROOT\CLSID\{C6A0FC60-3173-11D0-93D7-BA6083000000}\verb\0
    = &Edit,0,2
HKEY_CLASSES_ROOT\CLSID\{C6A0FC60-3173-11D0-93D7-
    BA6083000000}\InprocHandler32 = ole32.dll
```

Usually, when a finished application is installed, the Setup procedure also executes the REG script. For development purposes, you can also use the Registry Editor to execute this script directly. From the Registry menu in the Registry Editor, select Import Registry Files to open the file-selection dialog box. Select the application's REG script. A few seconds later, the Registry Editor should inform you that the registry information has been entered, which is all that is required.

Creating OLE Applications

In the past, creating any OLE application was a long and involved process requiring hundreds of lines of code, simply to provide the most rudimentary client capabilities. The good news is that creating an OLE client application using MFC and the AppWizard (or their equivalent in other compilers) is almost trivial. Just as you can use MFC and the AppWizard to create an OLE client, these services also provide the means to create a basic OLE server.

TIP

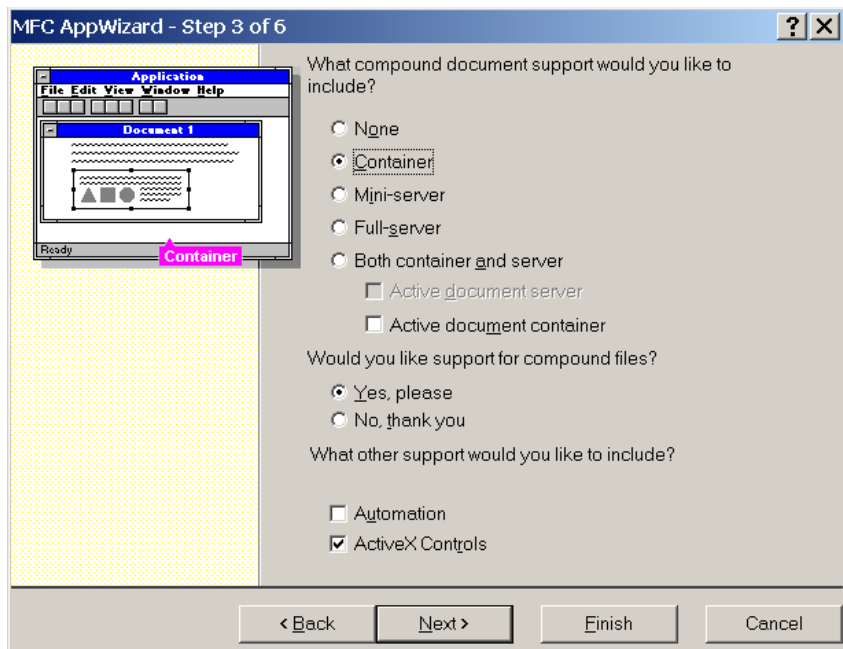
If you are interested in developing OLE applications, you should refer to any of the many books devoted to the topic. For example, *Mastering OLE 2* by Bryan Waters (published by Sybex) provides more details about how to provide extended OLE support in your applications.

The *OLE_Client* Demo: Creating an OLE Client Application

Using MFC's AppWizard to build the skeleton for your client application, in Step 3 of the AppWizard process, you are presented with an option to include OLE compound document support in your application, as shown in Figure S23.7. (In Step 1 of the AppWizard process, you must select a multi-document application; OLE support does not function for single-document applications.) By default, the None option is checked. To provide OLE client support, simply select the Container option from the list. Following Step 3, continue to specify the remainder of the options required for your application.

FIGURE S23.7:

Adding OLE client support
for an application

**WARNING**

While you are experimenting, there is one restriction to observe in creating an OLE client application. Do not name the application "OLE Client" or "OLEClient" (however, "OLE_Client", with an underscore, is permissible). Using either of these proscribed names results in a `COLEClientDoc` class being created as an application class, leading to a conflict with the library class of the same name, which is required to support OLE client operations.

When you complete the AppWizard specifications, MFC creates a multi-document interface with the usual object classes but includes one new object class: the client item, `COLE_ClientCntItem` (container item).

And—surprise!—you now have an OLE client application that is completely ready to compile, link, and execute. More important, the OLE client is ready to operate without any further provisions to support OLE. Granted, this simple application does not do much other than accept OLE support from server applications. However, given the complexity of creating an OLE client from scratch, this in itself is no small matter.

NOTE

The *OLE_Client* demo is included on the CD in the Supplement 23 folder.

Client Support and Control Methods

When you create an OLE client application through the AppWizard, it provides classes and functions that handle client support and connections to embedded or linked OLE items.

The `Cole_ClientView` Class

When you instruct AppWizard to provide OLE client support, the `Cole_ClientView` class is created, with seven OLE client-support functions, which are already fully implemented. These functions are discussed in the following sections.

OnInitialUpdate This function includes a provision to set the member variable `m_pSelection` to `NULL`, where the `m_pSelection` member is a pointer to a `Cole_ContainerCntrItem` object.

```
void Cole_ClientView::OnInitialUpdate()
{
    CView::OnInitialUpdate();
    m_pSelection = NULL;    // initialize selection
}
```

The default provision shown is adequate for most purposes. However, if you are going to use something besides the default selection mechanisms for variant server classes, you will need to provide the appropriate initialization.

OnDraw This function, which should be familiar from earlier examples, is expected to draw application-specific data for the client window (the document window). For OLE objects, this is also where the server drawing operations are implemented. Now, according to the stated purposes of OLE, the server application is responsible for doing the actual drawing, right? So, why do we need special provisions in the client's `OnDraw` function?

The reason is that before it can carry out its drawing instructions, the OLE server needs to know where to do the drawing operation.

```
void Cole_ClientView::OnDraw( CDC* pDC )
{
```

```

C0le_ClientDoc* pDoc = GetDocument();
ASSERT_VALID(pDoc);

// TODO: add draw code for native data here
// TODO: also draw all OLE items in the document

if( m_pSelection == NULL )
{
    POSITION pos = pDoc->GetStartPosition();
    m_pSelection =
        (C0le_ClientCntrItem*) pDoc->GetNextClientItem( pos );
}
if( m_pSelection != NULL )
    m_pSelection->Draw( pDC, CRect( 10, 10, 210, 210 ) );
}

```

In this default version, the selected OLE object is drawn at an arbitrary position using the rectangle returned by the `C0le_ClientCntrItem` class and an arbitrary drawing rectangle.

When you create a real application, your application must be responsible for positioning the OLE object and for determining the area appropriate for the object's drawing operations (see the description of the `OnSize` function).

IsSelected This method performs a test to determine whether a specific object corresponds to the `m_pSelection` object, returning either a `TRUE` or `FALSE` response.

```

BOOL C0le_ClientView::IsSelected( const CObject* pDocItem ) const
{
    // TODO: implement this to test for selected OLE client item
    return pDocItem == m_pSelection;
}

```

As long as the selection is limited to `C0le_ClientCntrItem` objects, no additional provisions are required. However, if you are planning to handle other types of selection mechanisms, this implementation will require revisions.

OnInsertObject This method serves two functions. The first function is to invoke the standard Insert Object dialog box to select an OLE object (described earlier in the supplement).

```

void C0le_ClientView::OnInsertObject()
{
    C0leInsertDialog dlg;
    if( dlg.DoModal() != IDOK ) return;
}

```

If the Insert Object dialog box does not return `IDOK`, no selection has been made and no further action is necessary. However, assuming that an OLE object has been selected, `OnInsertObject` is responsible for connecting the item to the application document. It begins by declaring a new instance of the `COle_ClientCntrItem` class.

```
BeginWaitCursor();
COle_ClientCntrItem* pItem = NULL;
TRY
{
    // create new item connected to this document
    COle_ClientDoc* pDoc = GetDocument();
    ASSERT_VALID( pDoc );
    pItem = new COle_ClientCntrItem( pDoc );
    ASSERT_VALID( pItem );
}
```

Next, the OLE item must be initialized from the dialog box data.

```
if( ! dlg.CreateItem( pItem ) )
    AfxThrowMemoryException(); // any exception will do
```

Assuming that the item was created from the class list (rather than from a file), the object's OLE server is launched to edit the item. In this case, however, editing is simply the method used to create the item's data rather than a process to change the object.

If there is a problem (because of a failure of the server or the system), an exception is thrown and the current TRY loop terminates before the CATCH response (following) takes over to report the failure.

NOTE

The term "throwing an exception" refers to intercepting an error condition and generating ("throwing") an exception condition to allow an exception handler to either correct the error or to recover from the error without terminating the application. See Chapter 9, "Exception Handling," for more information about handling exceptions.

Once the object has been created, the server is instructed to show the selected object.

```
ASSERT_VALID(pItem);
if( dlg.GetSelectionType() == COleInsertDialog::createNewItem )
    pItem->DoVerb( OLEIVERB_SHOW, this );
```

Notice that a number of `ASSERT_VALID` statements are included in the `TRY...CATCH` loop. These are present only for debugging purposes, and have no effect when the application is compiled for release. The `TRY...CATCH` loop, however, is active both in the debug version and the release version.

Last, the default provisions in the `OnInsertObject` method set the current selection (`m_pSelection`) to point to the last selected item before calling the document with an `UpdateAllViews` instruction, which will result in the document (and, therefore, the view) being refreshed.

```
    ASSERT_VALID( pItem );
    m_pSelection = pItem;    // set selection to last inserted item
    pDoc->UpdateAllViews( NULL );
}
```

If you don't want the last selected item to be the current selection, you can revise this portion of the code to create a different selection. However, regardless of how the default selection is made, clicking on an object in the document should still change the object selection.

The `CATCH` loop simply provides the standard error trapping when an error exception is generated.

```
CATCH( CException, e )
{
    if( pItem != NULL )
    {
        ASSERT_VALID( pItem );
        pItem->Delete();
    }
    AfxMessageBox( IDP_FAILED_TO_CREATE );
}
END_CATCH
EndWaitCursor();
}
```

OnCancelEditCntr This method provides the standard keyboard user interface (UI) to cancel an in-place editing session, allowing the client, not the server, to deactivate the operation.

```
void COle_ClientView::OnCancelEditCntr()
{
    // Close any in-place active item on this view.
    COleClientItem* pActiveItem =
        GetDocument()->GetInPlaceActiveItem( this );
```

```

        if( pActiveItem != NULL )
        {
            pActiveItem->Close();
        }
        ASSERT( GetDocument()->GetInPlaceActiveItem( this ) == NULL );
    }

```

OnSetFocus This method provides the special handling required for an object being edited in-place.

```

void COle_ClientView::OnSetFocus( CWnd* pOldWnd )
{
    COleClientItem* pActiveItem =
        GetDocument()->GetInPlaceActiveItem( this );
    if( pActiveItem != NULL &&
        pActiveItem->GetItemState() == COleClientItem::activeUIState )
    {
        // need to set focus to this item if it is in the same view
        CWnd* pWnd = pActiveItem->GetInPlaceWindow();
        if( pWnd != NULL )
        {
            pWnd->SetFocus();    // don't call the base class
            return;
        }
    }
    CView::OnSetFocus( pOldWnd );
}

```

The OnSetFocus method provided is used to check selection and set the focus to the appropriate OLE object. Except for very unusual circumstances, this method should not require revision.

OnSize This method allows the user to resize an OLE object by selecting the object and dragging the object outline.

```

void COle_ClientView::OnSize( UINT nType, int cx, int cy )
{
    CView::OnSize( nType, cx, cy );
    COleClientItem* pActiveItem =
        GetDocument()->GetInPlaceActiveItem( this );
    if( pActiveItem != NULL )
        pActiveItem->SetItemRects();
}

```

Except for very unusual circumstances, this method should not require revision.

The COle_ClientCtrItem Class

The COle_ClientCtrItem class is derived from the COleClientItem class and is used to provide a connection to an embedded or linked OLE item. The COle_ClientCtrItem class created by AppWizard is a minimal implementation; the real functionality is supplied by the parent COleClientItem class.

NOTE

There are more than 70 OLE-handling methods supplied by library and API functions. For more details, refer to the online documentation.

The COleClientItem creation methods provide functions to create both embedded and linked items from the clipboard services, from selected files, or by launching an OLE server. The IMPLEMENT_SERIAL macro generates the basic code required for a COBJEKT-derived class, providing runtime access to the class name and base class name defining the class position within the hierarchy. The constructor and destructor methods provided for the derived COle_ClientCtrItem class are skeletal and provide no functionality beyond the derived functionality of the parent.

```
IMPLEMENT_SERIAL( COle_ClientCtrItem, COleClientItem, 0 )
COle_ClientCtrItem::COle_ClientCtrItem( COleClientDoc* pContainer )
    : COleClientItem(pContainer)
{
    // TODO: add one-time construction code here
}

COle_ClientCtrItem::~COle_ClientCtrItem()
{
    // TODO: add cleanup code here
}
```

The default functionality should be sufficient for most purposes but, if you decide to add custom construction code to COle_ClientCtrItem, you need to include corresponding cleanup code in the destructor method.

OnChange Whenever an OLE item is being edited—whether the editing is occurring in-place or in a fully open server—OnChange notifications are sent to the client application to notify the client of changes in the state of the item or changes in the visual appearance of the content. The OnChange method allows the client application to update its own appearance.

```

void COle_ClientCtrItem::OnChange( OLE_NOTIFICATION nCode,
                                   DWORD dwParam )
{
    ASSERT_VALID(this);

    COleClientItem::OnChange( nCode, dwParam );
    // TODO: invalidate the item by calling UpdateAllViews
    //         (with hints appropriate to your application)
    GetDocument()->UpdateAllViews( NULL );
    // for now just update ALL views/no hints
}

```

Again, the default functionality will serve for most purposes. However, you may wish to alter the update performance for special circumstances.

OnChangeItemPosition This function is used during in-place activation to change the position of the in-place window. This may be done because changes to the data in the server document require a change in extent, or it may be a response to in-place resizing. The default operation is to call the base class `COleClientItem::OnChangeItemPosition` with the new in-place window rectangle. In turn, the `COleClientItem::SetItemRects` function is notified to move and/or resize the item to fit the specified rectangle.

```

BOOL COle_ClientCtrItem::OnChangeItemPosition( const CRect &cRectPos )
{
    ASSERT_VALID(this);
    if( ! COleClientItem::OnChangeItemPosition( cRectPos ) )
        return FALSE;
    // TODO: update any cache you may have of the item's
    //         rectangle/extent
    return TRUE;
}

```

If you wish to provide your own resizing implementation, refer to the `SetExtent` method for embedded OLE items.

OnGetItemPosition This method is called to determine the location of an item during in-place activation. The default implementation provided simply returns a hard-coded rectangle, which was defined by `AppWizard`.

```

void COle_ClientCtrItem::OnGetItemPosition( CRect &cRectPos )
{
    ASSERT_VALID(this);
    // TODO: return correct rectangle (in pixels) in cRectPos
}

```



```

        cRectPos.SetRect( 10, 10, 210, 210 );
    }

```

For a more sophisticated approach, this rectangle should reflect the current position of the item relative to the view used for activation. To obtain the view, call `COle_ClientCtrItem::GetActiveView`.

OnActivate This function is used to activate an OLE item in place by calling the `COleDocument::GetInPlaceActiveItem` function.

```

void COle_ClientCtrItem::OnActivate()
{
    COle_ClientView* pView = GetActiveView();
    ASSERT_VALID(pView);
    COleClientItem* pItem = GetDocument()->GetInPlaceActiveItem( pView );
    if( pItem != NULL && pItem != this )
        pItem->Close();
    COleClientItem::OnActivate();
}

```

NOTE

Only one item (per frame) can be activated at a time.

OnDeactivate This function is called when an item that was activated in place is to be deactivated. This restores the container application's user interface to its original state, hiding any menus and other controls that were created for in-place activation.

```

void COle_ClientCtrItem::OnDeactivateUI( BOOL bUndoable )
{
    COleClientItem::OnDeactivateUI( bUndoable );

    DWORD dwMisc = 0;
    m_lpObject->GetMiscStatus( GetDrawAspect(), &dwMisc );
    if( dwMisc & OLEMISC_INSIDEOUT )
        DoVerb( OLEIVERB_HIDE, NULL );
}

```

If `bUndoable` is `FALSE`, the container should disable the Undo command, in effect discarding the undo state of the container, because it indicates that the last operation performed by the server is not undoable.

Serialize This method is used to load or store data related to an OLE item within the client document. By default, the data contained within an OLE object is handled by the OLE server and does not require handling by the OLE client. Depending on the type of application you are designing, however, you may need to store references to the linked/embedded items as part of your document storage.

```
void COle_ClientCtrItem::Serialize( CArchive& ar )
{
    ASSERT_VALID( this );
    COleClientItem::Serialize( ar );

    // now store/retrieve data specific to COle_ClientCtrItem
    if( ar.IsStoring() )
    {
        // TODO: add storing code here
    }
    else
    {
        // TODO: add loading code here
    }
}
```

Other Methods There are a host of methods supplied by the parent class, `COleClientItem`, that may be overwritten when special handling is needed by your application. Status methods provide functions to retrieve OLE item aspects, including the item's class ID, the view aspect, and the OLE type and descriptive string.

The clipboard services support drag-and-drop operations and allow items to be retrieved from the clipboard or passed to the clipboard. Additional methods allow items to be drawn, closed, released, or executed. Object activation is provided by a series of functions handling different aspects of activation. The `SetExtent` and `SetItemRects` methods provide resizing. All in all, there are 20 or more functions for various aspects of OLE client operations.

Given these possibilities, you may be properly relieved to know that you do not need to write all of these yourself. For the most part, the default functionality for a client has been supplied. And, when necessary, you can override or extend the default methods.

Creating an OLE Server

Just as you can use MFC and the AppWizard to create an OLE client, these services also provide the means to create a basic OLE server, offering a choice of a mini-server or full-server application.

OLE Server Types

OLE server applications are defined by four base classes: the `COleServerDoc` and `COleServerItem` classes used by all server applications, the `COleServer` class used by mini-servers, and the `COleTemplateServer` class used by full-server applications.

A *Single Document Interface (SDI)* server is probably the most common type of OLE server, as well as the simplest to implement. Each SDI server uses a single server object and a single document object but launches a new server instance for each client requesting service. Table S23.1 shows the SDI architecture characteristics. Because mini-servers do not support multiple links, an SDI mini-server offers only one item object. In contrast, a full server supplies multiple item objects when multiple clients are linked to the same document.

TABLE S23.1: SDI Server, Multiple Instances

Class type	Classes	Mini-server objects	Full-server objects
Server	1	1	1
Document	1	1	1
Item	1	1	Many

Multiple Document Interface (MDI) servers are used when DGROUP (the default data segment) memory constraints preclude multiple-instance servers or when a full server needs to be MDI in stand-alone mode. For mini-servers, there is still only one item per document. Table S23.2 shows characteristics of the MDI server architecture for a single server type, single instance.

TABLE S23.2: MDI Server, Single Server Type, Single Instance

Class type	Classes	Mini-server objects	Full-server objects
Server	1	1	1
Document	1	Many	Many
Item	1	Many	Many

Multiple-instance MDI servers include applications such as Excel or Quattro Pro, which provide both charts (graphic objects) and spreadsheets. Each server class has only one document class, and each server object has one document object. Since full servers support links, each document can provide multiple item objects, and each document class can support multiple item classes. Table S23.3 shows the MDI server characteristics for multiple instances.

TABLE S23.3: MDI Server, Multiple Instances

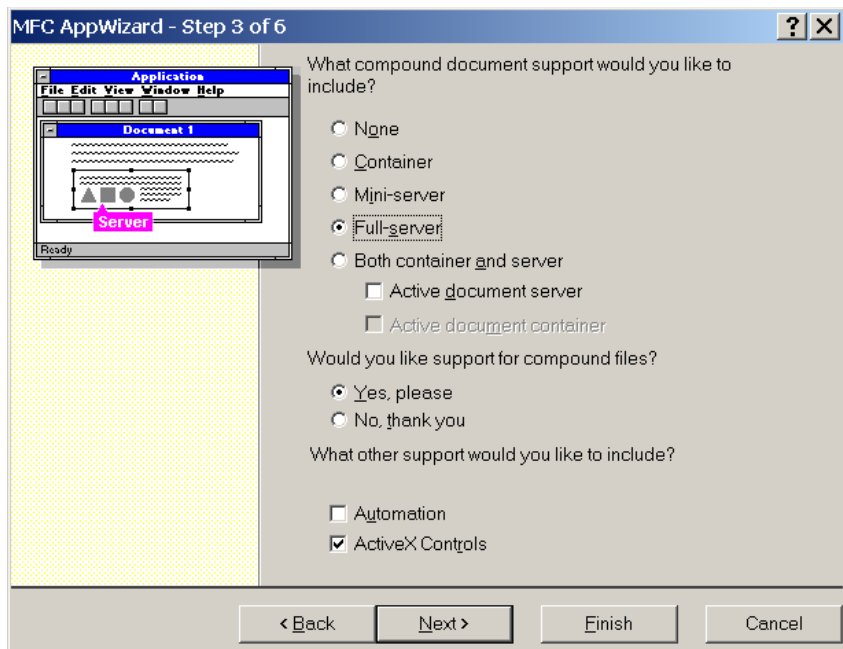
Class type	Classes	Full-server objects
Server	Many	Many
Document	Many	1
Item	Many	Many

Using the AppWizard

Just as you are presented with an option to include OLE support in your client application in Step 3 of the AppWizard process, this step also includes selections for a mini-server, full-server, or client/server application. In Figure S23.8, a full-server application has been selected.

FIGURE S23.8:

Adding OLE server support for an application



For OLE support, creating a mini-server and full-server application are essentially the same. However, it's easier to test a full-server application during development because it can also run in stand-alone mode. The combined client/server option automatically selects a full server rather than a mini-server, because the client side insists that the application must run in stand-alone mode (a mini-server cannot be a client without a user interface).

WARNING

When you are creating a full-server application, keep in mind that just because the server works correctly in stand-alone mode, this is no guarantee that it also works as a server. Later, we will discuss at least one point of failure where an application works by itself but fails during client operations.

After you have finished creating your application skeleton, in addition to the application, mainframe, document, and view classes, AppWizard has also created an in-place frame class, `CInPlaceFrame`, and a server class, `CxxNAMExxSrvrItem`,

derived from the `COleServerItem` class. For the *Parry* demo discussed in this supplement, the server class is named `CParrySrvrItem`.

The *Parry* Demo: A Simple OLE Application

The *Parry* demo is a relatively simple OLE application that is intended to demonstrate how an OLE application can provide embedded services. The embedded object offers a single menu option and a single toolbar option.

The menu and toolbar entries could be anything you want and could also duplicate the usual system menus, with File, Edit, and Help entries. Remember, however, that options provided by the OLE embedded menu and toolbar, which are presented in the client application when the embedded object is selected, must also be supported by the server application. Because the *Parry* demo does not offer File, Edit, or Help services in the context of the server application, no menu entries for these items have been provided.

The single service provided by the *Parry* demo is the Scan action, which, in true paranoid fashion, looks for enemies (and often finds them). No, this is not a serious application; it was designed with tongue firmly in cheek and for amusement as well as education. But even so, the principles demonstrated in the *Parry* demo still apply to serious server applications.

NOTE

The *Parry* demo is included on the CD, in the Supplement 23 folder.

Table S23.4 lists the main source files used in this demo. The following sections describe the classes used for the program.

TABLE S23.4: Principal Source Files in the *Parry* Demo

Source File(s)	Function
Program Files	
<code>SrvrItem.CPP, .H</code>	Source files for the <code>CParrySrvrItem</code> class
<code>MainFrm.CPP, .H</code>	Defines the <code>CMainFrame</code> class derived from <code>CMDIFrameWnd</code> ; controls all MDI frame features

Continued on next page

TABLE S23.4 CONTINUED: Principal Source Files in the *Parry* Demo

Source File(s)	Function
Resource Files	
ParryDoc.CPP, .H	Document class files for the server document class; modify these source files to add special document features and implement serialization
IpFrame.CPP, .H	Source files for the in-place frame class derived from the <code>COleIPFrameWnd</code> class; controls all frame features when the object is activated in-place
Parry.CPP, .H	Source files for the <code>CParryApp</code> application class
ParryView.CPP, .H	Source files for the view class files creating the <code>CParryView</code> class; handles in-place editing capabilities
IToolbar.BMP	Bitmap for in-place toolbar
Toolbar.BMP	Bitmap for stand-alone toolbar
Parry.ICO	Icon for stand-alone operation
ParryDoc.ICO	Icon for embedded object
Registry File	
Parry.REG	Registration script used to register the server

The CParrySrvrItem Class

The `CParrySrvrItem` class is derived from `COleServerItem` and provides the link functionality between the server application and the client through the OLE system. The parent class provides the default functionality, but there are ample opportunities within the derived class to customize the server's behavior.

The first point where the server can be customized is found in the constructor and destructor methods. The default versions are functional, but they can be modified to provide, for example, additional clipboard formats specific to the item's data source.

```
CParrySrvrItem::CParrySrvrItem( CParryDoc* pContainerDoc )
    : COleServerItem( pContainerDoc, TRUE )
{
    // TODO: add one-time construction code here
}
```

```

CParrySrvrItem::~~CParrySrvrItem()
{
    // TODO: add cleanup code here
}

```

Serialize The framework calls this method when a data item is copied to the clipboard, an action that happens automatically through the OLE callback function `OnGetClipboardData`.

The default provisions expect the server object to be embedded and delegate serialization to the document's `Serialize` function. Notice that the `IsLinkedItem` is called and expects a negative response (`FALSE`) to identify an embedded object.

```

void CParrySrvrItem::Serialize( CArchive& ar )
{
    if( ! IsLinkedItem() )
    {
        CParryDoc* pDoc = GetDocument();
        ASSERT_VALID( pDoc );
        pDoc->Serialize( ar );
    }
}

```

For linked support, additional provisions are needed to serialize only a portion of the server data.

OnGetExtent This method is designed to check the drawing aspect, returning a `CSize` variable with the appropriate size information.

```

// CParrySrvrItem::OnGetExtent is called to get the extent in
// HIMETRIC units of the entire item. The default implementation
// here simply returns a hard-coded number of units.

BOOL CParrySrvrItem::OnGetExtent( DVASPECT dwDrawAspect, CSize& rSize )
{
    if( dwDrawAspect != DVASPECT_CONTENT )
        return COleServerItem::OnGetExtent( dwDrawAspect, rSize );
}

```

If the drawing aspect is `DVASPECT_CONTENT`, the parent `OnGetExtent` method is called to retrieve the `rSize` variable. Otherwise, the default implementation provided by the `AppWizard` simply returns a hard-coded 3000 by 3000 units (in `MM_HIMETRIC` mode).

```

CParryDoc* pDoc = GetDocument();
ASSERT_VALID( pDoc );

```



```

        // TODO: replace this arbitrary size
        rSize = CSize( 3000, 3000 );    // 3000 x 3000 HIMETRIC units
        return TRUE;
    }

```

Normally, the server application is expected to handle drawing the content aspect of the item. To support other aspects, such as `DVASPECT_THUMBNAIL`, you need to override the `OnDrawEx` function and modify the `OnGetExtent` function to handle this additional aspect.

The drawing mode, identified by the `dwDrawAspect` parameter, may be `DVASPECT_CONTENT`, `DVASPECT_THUMBNAIL`, `DVASPECT_ICON`, or `DVASPECT_DOCPRINT`. To support any modes other than `DVASPECT_CONTENT`, additional provisions will be required both here and in the `OnDraw` method.

NOTE

Embedded or linked OLE items are always drawn using HIMETRIC units and a metafile device context. Of course, while the application is executing in stand-alone mode, any drawing mode is acceptable.

OnDraw This method is provided as a default implementation that sets up the mapping mode and extent in preparation for drawing in a metafile context. But, before you duplicate your entire application's drawing routines, realize that this is not the purpose of the server item's `OnDraw` method.

When the item is active, it is the `View` class that is called to provide the drawing operations. The function of the server item's `OnDraw` method is to act only when the OLE item is active, but the client screen still needs to be updated. It might provide instructions for a simple default drawing operation, for drawing an icon view, or for whatever is desired to provide an inactive display.

```

BOOL CParrySrvrItem::OnDraw( CDC* pDC, CSize& rSize )
{
    CParryDoc* pDoc = GetDocument();
    ASSERT_VALID( pDoc );

    // TODO: set mapping mode and extent
    // (the extent is usually same as size returned from OnGetExtent)
    pDC->SetMapMode( MM_ANISOTROPIC );
    pDC->SetWindowOrg( 0, 0 );
    pDC->SetWindowExt( 3000, 3000 );
    // TODO: add drawing code here. Optionally,
    // fill in HIMETRIC extent

```

```
        // all drawing takes place in the metafile device context (pDC)
        return TRUE;
    }
```

In addition to providing some form of drawing instructions here, the `SetWindowExt` function call should be rewritten to use the `CSize` value returned by the `OnGetExtent` function. Of course, this also assumes that the `OnGetExtent` function has been rewritten to return something besides the hard-coded values supplied by the `AppWizard`.

Also, if you want the OLE server view to be drawn both when the item is active and when it is inactive, rather than attempting to provide duplicate code for each operation, a simpler approach is to provide a set of shared drawing functions that are called for both.

OnDraw and Drawing in a Metafile Context Although the `OnDraw` methods of both the server item class and the view class are called with a pointer to a device context, the supplied device context is not the same in both cases. When the view's `OnDraw` function is called, the supplied device context is the screen device; when the server item's `OnDraw` method is invoked (while the item is inactive), the device context supplied is a metafile context.

There are several differences between a screen device context and a metafile context, but the difference you need to be most aware of when designing your OLE server is that a metafile context does not supply the same information as an active screen device context. The information supplied for a metafile context does not include any data that depends on the context actually being a window and an active element in the window hierarchy. This limitation means that functions such as `GetTextMetrics`, `GetDeviceCaps`, or many of the other `Getxxxxxxx` functions simply do not operate in a metafile context because there is no connection to any actual physical device context. In like fashion, functions such as `CreateCompatibleDC`, which might be used to create a bitmap memory context, simply do nothing.

In view of these limitations, the server item's `OnDraw` function must rely on `MM_ANISOTROPIC` mode with an extent defined in `MM_HIMETRIC` units. This selection is based on providing the highest resolution available.

This limitation means that both types of drawing operations must be carried out in `MM_HIMETRIC` units to make the normal drawing operations compatible with the inactive server drawing operations. Unfortunately, the `MM_HIMETRIC` mode is not necessarily ideal for this purpose. For example, consider that

application fonts must be rendered using `MM_HIMETRIC` units. (An alternative in this situation is to use conversion functions such as the `CFont::CreateFont` or `CFont::CreateFontIndirect` to force a font match based on the relative font size, instead of using a font that is sized according to the metafile context.)

Restrictions aside, most of the output functions, such as `MoveTo`, `TextOut`, and `DrawText`, still remain valid. Also, if necessary, the `CDC::HIMETRICtoDP` and `CDC::DPtoHIMETRIC` functions can be used to convert coordinates between the application's format context and device pixels.

The CParryView Class

The OLE application's view class, which is `CParryView` in this instance, provides the application view in both stand-alone mode and as an active OLE object. As usual, the `CParryView` class is derived from the `CView` class.

The AppWizard has included one provision in the `View` class to support server operations: the `OnCancelEditSrvr` method.

OnCancelEditSrvr This method parallels `OnCancelEditCntr` in the client application, providing the standard keyboard UI to cancel an in-place editing session. Here the method allows the server, not the client, to deactivate the operation.

```
void CParryView::OnCancelEditSrvr()
{
    GetDocument()->OnDeactivateUI(FALSE);
}
```

OnDraw Because the *Parry* application depends on a dialog box (or a series of dialog boxes) to present information as a pop-up service, the `View`'s `OnDraw` method really doesn't have much to do besides displaying a text string announcing its services.

```
void CParryView::OnDraw( CDC* pDC )
{
    CParryDoc* pDoc = GetDocument();
    ASSERT_VALID( pDoc );

    CString csText = "Paranoid Scanning Services";

    pDC->TextOut( 10, 10, csText );
}
```

However, if you have any graphics information to display (or if you have a more conventional display), this is where the drawing operations would occur, just as they would in any conventional application. Keep in mind, however, that the inactive display for an embedded server item is not drawn in the same context as the active display (see the “OnDraw and Drawing in a Metafile Context” section).

OnScan In the *Parry* demo, this method is called by the server-supplied menu or the server-supplied toolbar and uses a random-number generator to select the advice message to display. Technically, this is a very minimal application and shouldn’t require any particular explanation.

```
void CParryView::OnScan()
{
    CDialog *pDlg;
    UINT      nDlg;

    // Seed the random-number generator with current time so that
    // the numbers will be different every time we run.
    srand( (unsigned)time( NULL ) );
    switch( ( rand() % 10 ) + 1 )
    {
        case 1:  nDlg = IDD_DIALOG1;  break;
        case 2:  nDlg = IDD_DIALOG2;  break;
        case 3:  nDlg = IDD_DIALOG3;  break;
        case 4:  nDlg = IDD_DIALOG4;  break;
        case 5:  nDlg = IDD_DIALOG5;  break;
        default: nDlg = IDD_DIALOG6;  break;
    }
    if( nDlg != IDD_DIALOG6 )
        MessageBeep( MB_ICONEXCLAMATION );
    pDlg = new CDialog( nDlg, NULL );
    pDlg->DoModal();
    delete pDlg;
}
```

The CInPlaceFrame Class

The CInPlaceFrame class, derived from the COleIPFrame class, creates and positions the frame and control bars for the server window within the client application’s document window. The CInPlaceFrame class also handles notifications for embedded COleResizeBar objects whenever an in-place editing window is

resized. The parent class provides complete default functionality, but there are still possibilities for customization in the derived class.

The heart of the `CInPlaceFrame` class is found in two create functions: `OnCreate` and `OnCreateControlBars`.

OnCreate After calling the usual default method from the parent class, this method performs two tasks: setting up a `CResizeBar` instance to provide for in-place resizing, and providing a default drop target.

```
int CInPlaceFrame::OnCreate( LPCREATESTRUCT lpCreateStruct )
{
    if( COleIPFrameWnd::OnCreate( lpCreateStruct ) == -1 )
        return -1;
    if( ! m_wndResizeBar.Create( this ) )
    {
        TRACE0( "Failed to create resize bar\n" );
        return -1;    // fail to create
    }
    m_dropTarget.Register( this );
    return 0;
}
```

The default drop target does nothing for the frame window, but it does prevent drops (as in “drag-and-drop” operations) from falling through to another class that does support drag-and-drop, such as the OLE client.

If your application will be supporting drop operations, then this registration will be necessary anyway—along with provisions to handle drops, naturally.

OnCreateControlBars This method will be called, as required, by the framework to create control bars for the container application’s windows.

```
BOOL CInPlaceFrame::OnCreateControlBars( CFrameWnd* pWndFrame,
                                         CFrameWnd* pWndDoc )
{
    // set owner to this window, so messages are delivered to correct
    // app
    m_wndToolBar.SetOwner( this );
    // create toolbar on client's frame window
    if( ! m_wndToolBar.Create( pWndFrame ) ||
        ! m_wndToolBar.LoadToolBar(IDR_SRVR_INPLACE) )
    {
        TRACE0( "Failed to create toolbar\n" );
        return FALSE;
    }
}
```

The `pWndFrame` argument is the client application's top-level frame window and is always non-NULL. The `pWndDoc` argument is the document-level frame window and, if the client is an SDI application, may be NULL. The server application may place control bars on either window, but in this case, the principal task is to create the toolbar and (by default) to dock the toolbar to the client's document window.

```
// TODO: remove this if you don't want tool tips
//      or a resizable toolbar
m_wndToolBar.SetBarStyle( m_wndToolBar.GetBarStyle() | CBRS_TOOLTIPS |
                          CBRS_FLYBY | CBRS_SIZE_DYNAMIC );
// TODO: delete these three lines if you don't want the toolbar to
//      be dockable
m_wndToolBar.EnableDocking( CBRS_ALIGN_ANY );
pWndFrame->EnableDocking( CBRS_ALIGN_ANY );
pWndFrame->DockControlBar( &m_wndToolBar );
return TRUE;
}
```

The toolbar docking and tool tips provisions, as well as the toolbar button assignments, provide operations only while the OLE item is active in-place. If the OLE item is not active, the toolbar (and menu) will not appear.

The CParryApp Class

The `CParryApp` class is derived from the `CWinApp` class. In earlier examples, we have paid little attention to any of the applications' `CWinApp`-derived classes. We've simply assumed that these classes were there and that they provided, by default, the essentials necessary for initializing and executing our application instances. For our OLE server application, however, the derived `CParryApp` class continues to supply the same functionality as previous examples, but now also offers a few elements that non-OLE applications haven't needed. These elements are described in the following sections.

The CLSID Value This is the CLSID (CLasS ID) value used by the system registry. The CLSID value is defined in the `CParry.CPP` file as:

```
static const CLSID clsid =
{ 0xc6a0fc60, 0x3173, 0x11d0,
  { 0x93, 0xd7, 0xba, 0x60, 0x83, 0x0, 0x0, 0x0 } };
```

The generated value, `C6A0FC60-3173-11D0-93D7-BA6083000000`, is statistically unique but may be changed if you want to substitute another identifier.

InitInstance This function begins by initializing the OLE libraries through a call to `AfxOleInit`, reporting failure if there is an error.

```

BOOL CParryApp::InitInstance()
{
    // initialize OLE libraries
    if( ! AfxOleInit() )
    {
        AfxMessageBox( IDP_OLE_INIT_FAILED );
        return FALSE;
    }
}

```

Next, as usual, the standard profile settings are loaded, and the document, window, and view class names are assigned.

```

LoadStdProfileSettings();
CSingleDocTemplate* pDocTemplate;
pDocTemplate = new CSingleDocTemplate( IDR_MAINFRAME,
                                     RUNTIME_CLASS(CParryDoc),
                                     RUNTIME_CLASS(CMainFrame), // main SDI frame
                                     // window
                                     RUNTIME_CLASS(CParryView) );

```

Now the `InitInstance` routine departs from the usual routine by calling the `SetServerInfo` function to identify server resources, including menus and accelerator tables, which are used by the server application when an embedded object is activated.

```

pDocTemplate->SetServerInfo( IDR_SRVR_EMBEDDED, IDR_SRVR_INPLACE,
                           RUNTIME_CLASS(CInPlaceFrame) );
AddDocTemplate( pDocTemplate );

```

The `ConnectTemplate` function is called to connect the server to the document template so that `COleTemplateServer` can use information in the document template to create new documents on behalf of OLE clients.

```

m_server.ConnectTemplate( clsid, pDocTemplate, TRUE );

// note: SDI applications register server objects only
// if /Embedding or /Automation is present on the
// command line.

// parse command line for standard shell commands, DDE, file open
CCommandLineInfo cmdInfo;
ParseCommandLine(cmdInfo);

```

Finally, after parsing any command-line instructions, a check is made to determine if the application instance is being launched as an OLE server and, if so, to register all of the OLE services as running (`RegisterAll`), allowing the OLE libraries to create objects from other applications.

```
if( cmdInfo.m_bRunEmbedded || cmdInfo.m_bRunAutomated )
{
    COleTemplateServer::RegisterAll();
    return TRUE;
}
```

If the application is being executed as a server rather than as a stand-alone application, the `InitInstance` routine returns `TRUE` here, so that the application's main window is not created or displayed.

On the other hand, if the application is being run as a stand-alone application, this is a good time to call `UpdateRegistry` to update the system registry with information about the OLE services before proceeding with normal operations (including starting the command processor and displaying the application window).

```
m_server.UpdateRegistry( OAT_INPLACE_SERVER );
if( ! ProcessShellCommand(cmdInfo) )
    return FALSE;
return TRUE;
}
```

Summary

Programming OLE client/server applications is a major topic deserving much more than a single chapter, but at least we have given you an introduction and an overview of OLE requirements. In particular, we've started by taking a look at the basics of OLE operations and the OLE libraries before discussing OLE server and registration. Finally, after briefly studying developing client and server applications, we finished with *Parry*, a relatively simple but paranoid OLE server.

A P P E N D I X

A

A

Windows 2000 Certification

- Windows 2000 Readiness Program
- Rational Windows 2000 TestFoundation
- Authoring Tools for Windows Installer Service

As Windows 2000 becomes the newest operating system in Microsoft's arsenal, a set of certification standards for Windows 2000 applications has also been produced. To qualify for Windows 2000 certification, an application must:

- Comply with the Application Specification for Windows 2000.
- Undergo compliance testing by VeriTest.
- Be able to run on any of the Windows 2000 operating system versions: Windows 2000 Professional, Windows 2000 Server, Windows 2000 Advanced Server, or Windows 2000 Datacenter.
- Optionally, developers can also choose to have applications certified on additional Windows operating systems.

The Certified for Windows logo issued for an application will indicate the operating system(s) that the application has been certified for.

NOTE

Information on the Windows 2000 certification program can be found at msdn.microsoft.com/certification/default.asp.

Additional resources for development and certification are available, as described in the following sections.

Windows 2000 Readiness Program

The Windows 2000 Readiness Program is a free program from Microsoft offering such technical benefits as free developer support and training opportunities to help developers meet the Application Specification for Windows 2000.

NOTE

Information on the Windows 2000 Readiness Program can be found at msdnisv.microsoft.com/msdnisv/win2000/.

Rational Windows 2000 TestFoundation

The Rational Windows 2000 TestFoundation is a free downloadable set of test tools, including common tools, data, methods, and metrics created by Rational Software to help developers introduce compliance testing early on in their application development life cycle. To provide automated testing solutions, the Rational Windows 2000 TestFoundation can be combined with Rational TeamTest.

Authoring Tools for Windows Installer Service

Two installation authoring services are available: InstallShield for Windows Installer (see Appendix B) and Wise for Windows Installer. Both allow setup authors to build installations that use the Microsoft Windows installer service—a requirement for Windows Certification. Both vendors provide tools and top-level technical support to software developers creating installations for Windows 2000 applications.

NOTE

Information about InstallShield for Windows Installer can be found at www.installshield.com/iswi/.

NOTE

Information about Wise for Windows Installer can be found at www.wisesolutions.com/wfwi/.

A P P E N D I X

B

B

Windows InstallShield

- How InstallShield Operates
- A Shortcut to Creating a Setup Project
- Finishing the Setup Project
- InstallShield's Script Structure
- Building a Disk Image
- Limits and Shortcomings

Application installation has always been a special headache, both for the application developer and for the prospective application users. Historically, installation solutions have varied from complex utility scripts prompting users for drive and directory information to equally complex instructions intended to guide users through confusing choices, options, and requirements.

Functionally, few (if any) of these “solutions” have proved exceptionally satisfactory and the need for a “better mousetrap” has been more than evident for most of the history of the personal computer.

Today, as a very different solution, InstallShield for Microsoft Windows (distributed with Visual Studio 6.0) provides the means for creating a complete, semi-automatic installation package for distributing both commercial and in-house applications. InstallShield installation packages are customized for individual applications and can not only handle file transfer operations but also provide registry entries and other special provisions.

In operation, InstallShield provides its own IDE together with guides to defining installation requirements—wizards, in effect, to walk you through the creation of a complete setup operation. The completed executable setups include provisions for uninstalling applications, for making entries in the Start Programs menu and for building disk images that are appropriate for the desired distribution media.

Using InstallShield, a complete installation package, including uninstall provisions, can be created in a matter of minutes. Of course, exactly how much time is required will depend on the complexity of your application package and how ready your components are for packaging.

TIP

While InstallShield for Microsoft Visual C++ 6 is a powerful 32-bit setup creation application, it is also only a subset of the full features and power available in InstallShield 5.1, Professional Edition.

How InstallShield Operates

InstallShield guides you through designing an application setup by organizing applications as building blocks consisting of application files, file groups, components, and not least, setup types.

As a first step, application files are bundled into file groups that are then linked into components and subcomponents. Finally, components are associated (as appropriate) with setup types.

Once this is done, for installation, users simply select the components or subcomponents they wish to install without needing to worry about which specific files they need to install or the directory structures needed.

Grouping Files

One of the primary tasks in setting up any application is transferring files from the distribution media to the target system (nominally, but not always, a hard drive). Normally, files can be grouped according to function, usage, or installation location. For example, file groups could be application executables, data files, help files and documentation, or system DLLs. Frequently, these groups may also correspond to intended destination directories, but regardless of directory structures, any selection of files that have common characteristics can be defined as a file group.

Other characteristics that could be applied to define groups are compressed/uncompressed, international languages, locked/not locked, purpose/function, shared/not shared, or supported operating system or system version. These groups do not necessarily need to correspond to specific application installation configurations but can simply be how you (as the setup author) view the application files.

Component Organization

While file groups can represent the developer's view of the application, components and subcomponents should represent the client's (user's) view of the application. That is, component organization is used to produce functional groups that—unlike file groups—do mirror installation configurations.

If your setup is intended for a single application, then you will have only one component group. Alternately, if your application is distributed in several versions—for example, for different operating system versions—then each version could comprise a separate component. Or, of course, if your setup is for a product suite, each separate product can be the basis for a separate component. For example, office suites such as Corel's or Microsoft's are comprised of multiple applications including a word processor, spreadsheet, presentation manager, and database, among others.

While all of these applications are distributed in a single package, individual users may wish to install only specific programs. Organizing these as components for setup makes it possible to select which elements will be installed and which will not.

Further, each component can consist of or include subcomponents representing optional utilities, tutorial programs, support files, or supplementary data files.

Setup Types

While components provide a means for organizing selective installations, setup types offer a different level of convenience. Simply offering the user access to select or reject components for a custom installation is fine, but providing setup types—predefined configuration sets—offers the convenience of not having to work through complex choices. For example, offering the user a choice of the setup types Normal, Minimal (for portable systems), and Custom allows the first two types to satisfy 90% of the user installations, while the third (Custom) would allow users with special requirements to tailor their installations however they wish.

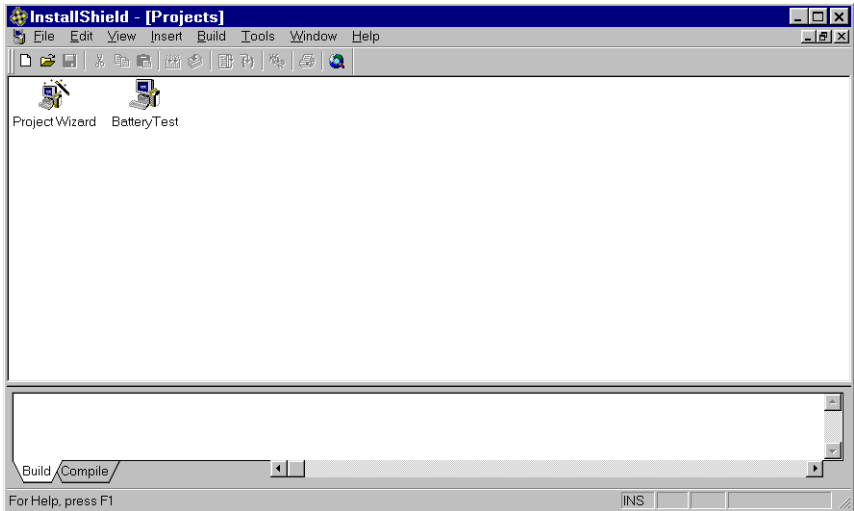
In this fashion, most users are saved from an unnecessary maze of decisions, while those who do have custom needs are still free to pick and choose among options.

Creating an Application Setup

The first step in creating an application setup using InstallShield is to take advantage of InstallShield's Project Wizard. After launching InstallShield (from the VC++ Tools menu), select the Project Wizard icon (Figure B.1).

The Project Wizard can help you create an executable setup in fifteen minutes or less. The Project Wizard creates a setup script that includes IDE settings for the operating system, language selections, file groups, Components, and setup types. Once created, the setup script can be modified as necessary.

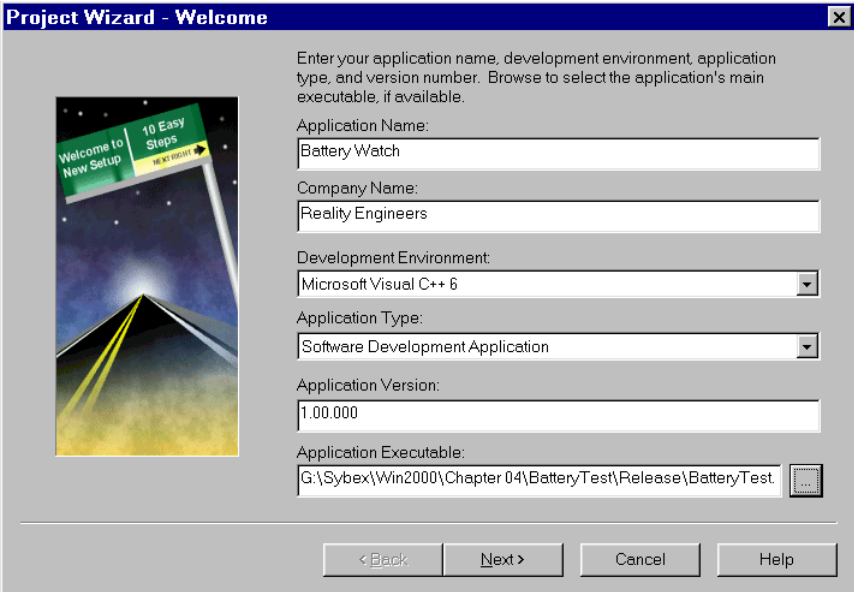
FIGURE B.1:
Starting the InstallShield
Project Wizard



Basic Project Information

Initially, the Project Wizard will request some basic information about the application (see Figure B.2).

FIGURE B.2:
Application Information



The most essential items, of course, are the application name and the application executable—in this example, the *BatteryTest* demo from Chapter 4.

Setup Dialogs

In the next step, the Project Wizard offers a series of dialog boxes that can be included in the setup application. These consist of:

Welcome Message *

Software License Agreement *

Readme Information

User Information *

Choose Destination Location *

Setup Type *

Custom Setup *

Select Program Folder *

Start Copying Files

Setup Complete *

Asterisks (*) show dialogs that are selected by default, but the list can be modified as desired.

For the demo installation for the *BatteryTest* program, only the four dialogs listed in **boldface** are used. This selection avoids displaying a license agreement (since there is none), omits requesting user information and a serial/registration number (unnecessary, see “Limits and Shortcomings”) and skips requesting a setup type (there’s only one). Instead, the resulting product simply asks for a destination location and program folder and performs the installation—which is all that *BatteryTest* really requires.

Operating Systems

The Operating Systems screen in the Project Wizard offers a choice of showing only the available platforms (that is, the platforms for which you have purchased setup availability from InstallShield Corporation—in other words, the WinTel x86 platforms) or all platform types (includes WinNT MIPS and Alpha platforms).

From the list, select the operating system(s) on which the application is designed to perform.

WARNING

Since 16-bit platforms such as Windows 3.1 do not support long filenames, if you are targeting a 16-bit platform, avoid long filenames for application files.

In this example, the selected targets are Windows 95 and NT 3.51/4.0. (No options for Windows 98 or Windows 2000 are included at present.)

Supported Languages

The Specify Languages screen offers a list of available languages. Select those languages that your application will be localized to support.

Setup Types

Project Wizard offers a list of seven basic setup types: Compact, Typical, Custom, Network, Administrator, Network (Best Performance), and Network (Efficient Space). Two or more setup types can be selected and each can be modified during setup creation.

Alternately, if you do not choose to offer a choice of installation types—as, for example, when you only have one installation option—click on a blank entry to deselect all types.

Component List

The Components list offers four basic component types: Program Files, Example Files, Help Files, and Shared DLLs. Existing component types can be deleted from the list or additional component types can be added as needed. All components will be modifiable at any point during setup creation.

File Groups

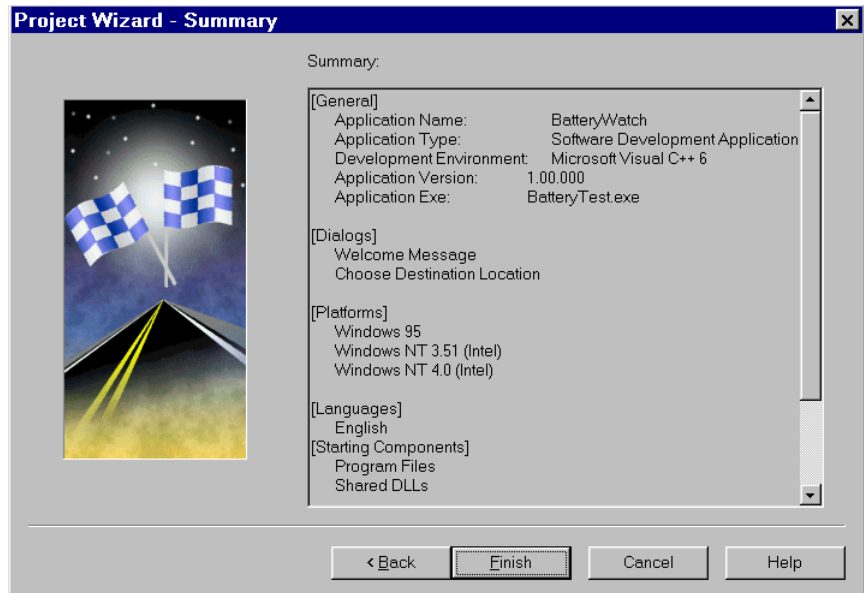
File groups are used to create logical sets of files that comprise an application. A default set of five file groups is supplied: Program Executable Files, Program DLLs, Example Files, Help Files, and Shared DLLs. (Note the close correspondence between the file groups and the components list.)

Summary Panel

The Summary panel (see Figure B.3) shows the choices and settings made thus far.

FIGURE B.3:

Project Summary



Click the Finish button, and the Project Wizard will create the setup project and open the Project workspace shown in Figure B.4.

A Shortcut to Creating a Setup Project

A shortcut to creating a setup project is also available. If you have your application project open in Visual Studio (VC/C++), before you call InstallShield from the Tools menu, you can simply select the project directly and skip many of the preceding steps.

The downside, however, is that a variety of default settings are used and that you have less freedom to customize the installation process. While the results are still functional, the choices may or may not be entirely appropriate to your project's needs. The defaults will, however, automatically include both local and system DLLs required for support as well as the application's executable files.

In either case, whether the shortcut or the full process is used, the finishing steps (following) will still be required.

Finishing the Setup Project

At this point, InstallShield's Project Wizard has created a basic script for the setup project. There are, however, a few additional steps before the setup is complete:

- Assigning files to file groups
- Assigning file groups to component groups
- Adding the "your application" icon to the Start Programs menu
- Creating a disk image

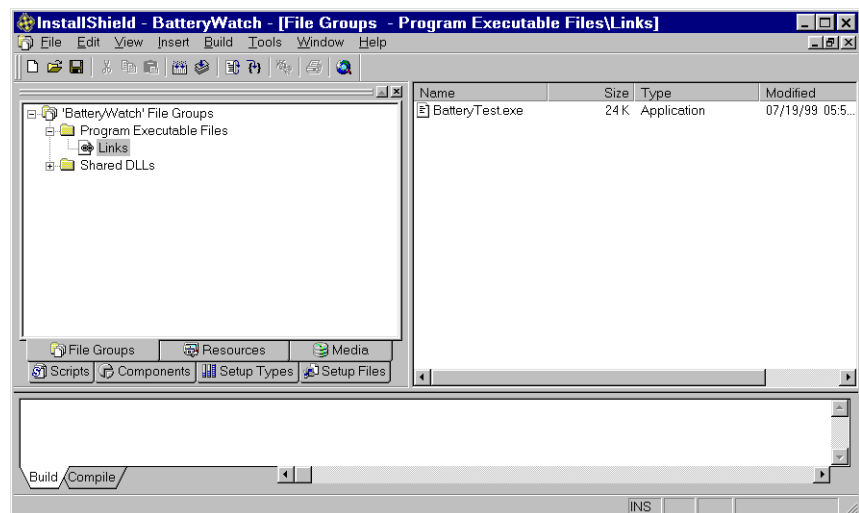
Assigning Files to File Groups

To create the BatteryWatch setup, the BatteryTest.exe program needs to be added to the Program Executable Files file group:

1. Select the File Groups pane tab (Figure B.4), then opening the Program Executable Files folder icon and the Links subfolder.

FIGURE B.4:

Assigning Files to File Groups



2. In the File Group Links window, a right-click with the mouse brings up a pop-up menu. From the menu, select Insert Files to open the Insert File Links to FileGroup dialog box.
3. Use the file selection dialog box to locate and select all executable files that will be installed as part of this project. For the demo application, the only executable file is `BatteryTest.exe`. After selection, the files (and directory information) appear in the File Group Links window.

In the example, since the *BatteryTest* application relies on two system DLLs—`MFC42.dll` and `MSVCRT.dll`—these are added to the Shared DLLs link in the same fashion.

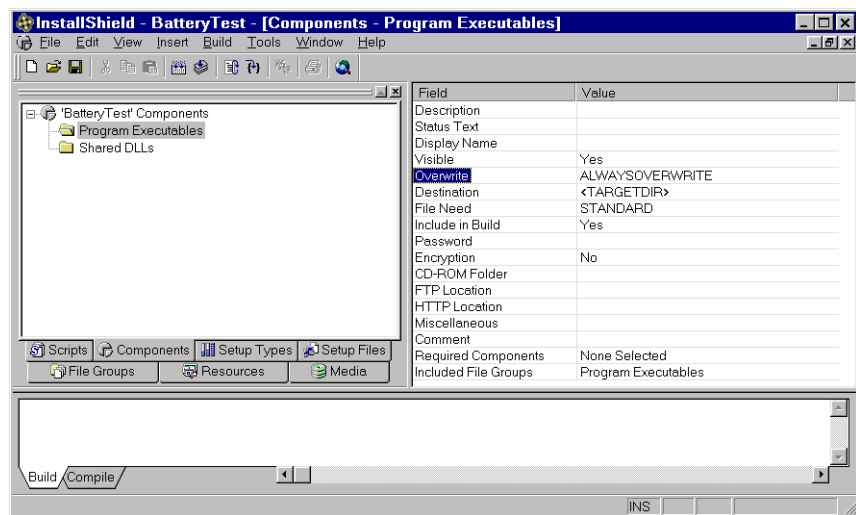
Assigning File Groups to Components

The next step is to assign files—from the Program Executable Files and the Shared DLLs file groups—to Program Executable and Shared DLLs components. The process is similar to assigning files to file groups:

1. Select the Components pane tab and open the Program Files folder icon to show the Component Properties window (see Figure B.5).

FIGURE B.5:

The Component—Program Executables window



2. In the Component Properties window, double-click on the Included File Groups item (at the bottom of the list) to open the Included File Groups property page.
3. Click the Add button to open the Add File Group dialog box and select Program Executable Files from the File Group Name list.

Repeat as necessary for additional components—like, in the example, the Shared DLLs component.

The properties for each component group can be modified as necessary, but the default settings will serve adequately in most cases. When changes or modifications are necessary, simply double-click on any item in the Component Properties to bring up a dialog box showing the options and offering an explanation for the property.

Adding the Application Icon to the Start Programs Menu

Adding the application's icon to the Start Programs menu requires changes to the setup script. Begin by selecting the Scripts tab. In return, the `Setup.rul` file should be visible in the Script Editor window. If it is not, select `Setup.rul` from the Scripts files tree.

Next, scroll through the script to find the `SetupFolders()` function or use the Find... function from the Edit menu to search for "function `SetupFolders`." The default function code should look something like this:

```
function SetupFolders()
    NUMBER nResult;

begin
    // TODO : Add all your folder (program group) along with
    // shortcuts (program items)
    //
    //     CreateProgramFolder, AddFolderIcon....
    //
    nResult = CreateShellObjects( " " );
    return nResult;
end;
```

Notice that the script code looks something like a crossbreed between C/C++ and Pascal, or maybe with elements of Visual Basic thrown in. In one sense, it is

none of the above—but also appears to have elements of all of these. Still, the programming required is relatively simple and online help is readily available.

The next couple of steps, for those who are unfamiliar with InstallShield's script language, require some explanation, not because the programming is particularly complex but simply because the tools and forms are unfamiliar.

The objective is to add code to the script that will place the application's icon on the Start Programs menu. This begins by declaring a string variable and then setting the variable equal to the application path before passing the variable to the `LongPathToQuote` and `AddFolderIcon` functions.

First, add the variable declaration to the function header, preceding the `begin` keyword:

```
STRING svPath;
```

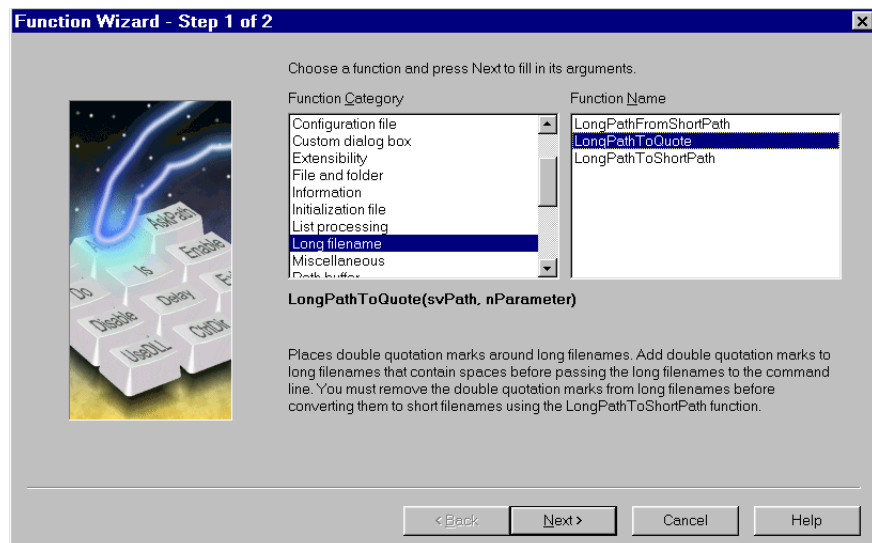
Next, following the `TODO` comment block, add a command to set the variable (`svPath`) equal to the path to the application:

```
svPath = TARGETDIR ^ "Notepad.exe";
```

On a blank line following this entry, right-click on the line to call a pop-up menu and from the menu select `Function Wizard`. The `Function Wizard` allows function calls to be pasted into the script and appears in Figure B.6.

FIGURE B.6:

The Function Wizard



In the Function Wizard, select Long filename in the category list and in the Function Name list select LongPathToQuote. This places double quotation marks around long filenames as required by the AddFolderIcon function.

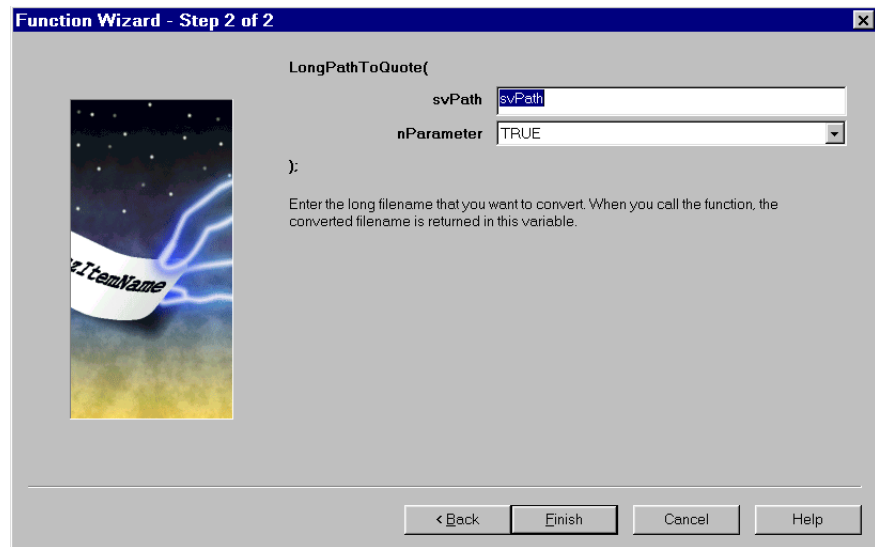
TIP

The Function Wizard's Help button can be used to learn more about any of the functions.

Select the Next button to open step 2 (see Figure B.7) in the Function Wizard.

FIGURE B.7:

Modifying Function
Parameters



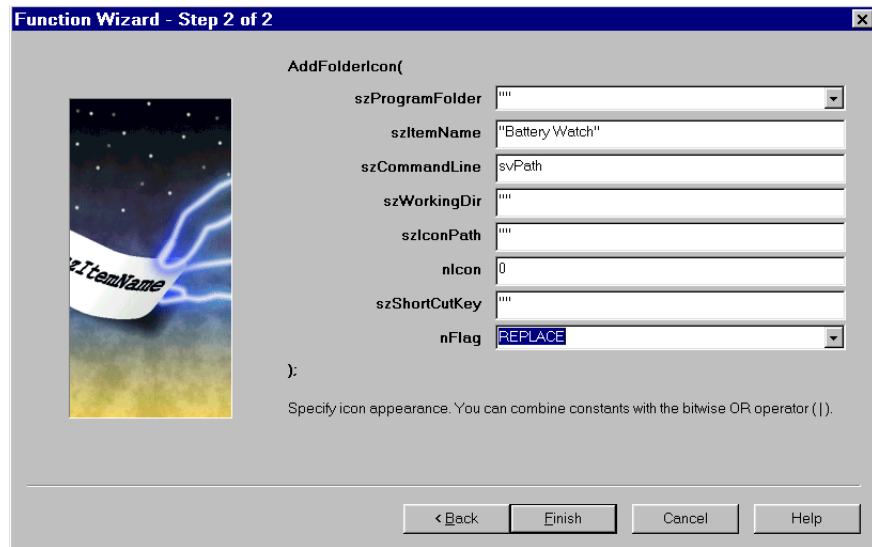
In this case, since the default parameters are fine and no modifications are required, simply click the Finish button to paste the function into the script.

The next function to add to the script requires a bit more work. If necessary, add new lines to the script, then right-click to bring up the menu, and again select Function Wizard.

This time, select Shell from the Function Category list and then select AddFolderIcon from the Function Name list. This is the function that will add an icon to the specified folder.

Click Next to continue with step 2 of the Function Wizard (see Figure B.8) and modify the parameter list of the `AddFolderIcon` function.

FIGURE B.8:
Modifying the Parameter List



Function Wizard - Step 2 of 2

`AddFolderIcon(`

szProgramFolder: ""

szItemName: "Battery Watch"

szCommandLine: svPath

szWorkingDir: ""

szIconPath: ""

nIcon: 0

szShortCutKey: ""

nFlag: REPLACE

`);`

Specify icon appearance. You can combine constants with the bitwise OR operator (|).

< Back Finish Cancel Help

While the Function Wizard has supplied arguments for the parameters, these defaults are not appropriate for the present task and the supplied entries have been changed, as shown in Figure B.8.

TIP

The blank strings—shown by quotation marks ("")—are required entries.

Click the Help button to display an explanation for the function and parameter requirements. The `nIcon` entry, for example, is specified as 0, identifying the first icon in the executable application.

After changing the parameter list, click Finish to paste the completed function into the script.

When you are finished, the `SetupFolders` function should look something like this:

```
function SetupFolders()
    NUMBER nResult;
    STRING svPath;
```

```

begin
    // TODO : Add all your folder (program group) along with
    // shortcuts (program items)
    //
    //     CreateProgramFolder, AddFolderIcon....
    //
    svPath = TARGETDIR ^ "BatteryTest.exe";
    LongPathToQuote ( svPath , TRUE );
    AddFolderIcon ( "" , "Battery Watch" , svPath , "" ,
                  "" , 0 , "" , REPLACE );
    nResult = CreateShellObjects( "" );
    return nResult;
end;

```

Finally, select Save from the File menu to save the setup project.

TIP

Unlike Visual Basic, line breaks within function definitions (see the `AddFolderIcon` function above) are permitted. Instead, like C/C++ or Pascal, a semicolon must be used to terminate each instruction.

InstallShield's Script Structure

The structure of an InstallShield script, while not precisely like any other language, is similar to C/C++, Pascal, or Visual Basic and you should be able to puzzle out most of the requirements simply by reading the generated script supplied by InstallShield's Wizard. However, as a brief overview, the script begins with `#include` statements—like those used in C/C++—for header files as:

```

// Include header file
#include "sdlang.h"
#include "sddialog.h"

```

Following the header declaration, the string defines also follow C/C++ format as:

```

////////// string defines //////////

#define UNINST_LOGFILE_NAME      "Uninst.isu"

```

The next block—installation declarations—offers a location to define any DLL prototypes that you may be using to customize the installation process. Since the *BatteryTest* demo is quite simple, no custom DLLs are invoked during installation, and obviously no prototypes are needed.

```
////////// installation declarations //////////
```

```
// --- DLL prototypes ---
```

```
// your DLL prototypes
```

The next block in the code consists of function prototypes:

```
// -- script prototypes ---
```

```
// generated
prototype ShowDialogs();
prototype MoveFileData();
prototype HandleMoveDataError( NUMBER );
...
```

Notice that the functions do not specify a return type although they may identify a parameter type. The generated script contains a number of prototypes for functions that have been supplied by the Wizard. Any custom functions that you choose to add to the setup script would also require prototype definitions following the same format. Following the function prototypes, global variables are defined:

```
BOOL    bWinNT, bIsShellExplorer, bInstallAborted,
        bIs32BitSetup;
STRING  svDir;
STRING  svName, svCompany, svSerial;
STRING  szAppPath;
STRING  svSetupType;
...
```

Here again, the format follows C/C++ conventions and should be relatively familiar. Any custom (global) variables required by your installation would follow the Wizard-supplied variables.

The main program, identified by the program label, consists of a linear process. Remember that this is an installation script, not an application, and, since certain tasks are expected to be performed in sequence (once only), there really isn't any

need for elaborate branching or any special provisions to control the order of execution.

```

program
  Disable( BACKGROUND );
  CheckRequirements();
  SetupInstall();
  SetupScreen();
  if( ShowDialogs() < 0 )      goto end_install;
  if( ProcessBeforeDataMove() < 0 ) goto end_install;
  if( MoveFileData() < 0 )    goto end_install;
  if( ProcessAfterDataMove() < 0 ) goto end_install;
  if( SetupRegistry() < 0 )   goto end_install;
  if( SetupFolders() < 0 )    goto end_install;

```

Notice that the first several functions called are not tested—the assumption is simply that these will perform, but any results returned are irrelevant. Granted, this does seem a little odd in the case of the `CheckRequirements` function, but examination of the function proper will show that the function uses an `Abort` operation, presumably to terminate the script if the system requirements aren't satisfied.

In any case, the last six procedures are tested for an error result (some negative value) and branch to the `end_install` label if an error occurs. The `end_install` label is a small curiosity of its own, since it is identified by a terminal colon (:) rather than a leading colon.

```

end_install:
  CleanupInstall();
  // If an unrecoverable error occurred, clean up the
  // partial installation. Otherwise, exit normally.
  if( bInstallAborted ) then
    abort;
  endif;
endprogram

```

Also notice that this `if` statement uses an `endif` while the previous `if` statements—which branched to a label—did not.

In any case, the overall structure of the program's main routine remains essentially linear. For the most part, the subroutines are also linear tasks, although some of the subroutines can be less linear. The `ShowDialogs` subroutine, for

example, relies on a series of labels and `if` statements to potentially loop back after each call to a subroutine.

```
function ShowDialogs()
    NUMBER nResult;
begin
   Dlg_Start:      // beginning of dialogs label

   Dlg_SdWelcome:
        nResult = DialogShowSdWelcome();
        if( nResult = BACK ) goto Dlg_Start;

   Dlg_SdAskDestPath:
        nResult = DialogShowSdAskDestPath();
        if( nResult = BACK ) goto Dlg_SdWelcome;

   Dlg_SdSelectFolder:
        nResult = DialogShowSdSelectFolder();
        if( nResult = BACK ) goto Dlg_SdAskDestPath;

    return 0;
end;
```

In this particular subroutine, each of the called functions displays a dialog with [Next] and [Back] buttons, and if the [Back] button is selected, the subroutine returns to the previous step.

Subroutine Structures

Like other elements in the InstallShield script, the structure of a subroutine is both familiar and quite different from other programming languages.

First, the term `function` is required to identify the name as a function label. Next, the label is followed by any local variable declarations before the keyword `begin` identifies the start of the actual function.

```
function FunctionName()
    NUMBER nResult;
begin
```

The body of the function can consist of a variety of operations and there's really no purpose in trying to delineate all of the possibilities here. You should note, however, that a `switch/case` structure—similar to C/C++—is commonly used for branching.

Also, without trying to examine all of the possibilities in detail, it appears likely that any complex operation—such as verifying an encoded registration key—might be easier to implement as a function supplied by an external DLL than as a direct part of the script.

Regardless of the internal operations, each function has the potential of a LONG value as a response code. Any other data types needed could be returned as parameters, declared using a BYREF statement (similar to Visual Basic).

```
    return nResult;  
end;
```

Finally, each function is terminated by an end statement.

TIP

All individual statements are terminated—C/C++ fashion—with a semicolon.

Functions that are called with arguments contain only the argument name (local variable name) while the argument types have been previously declared in the function prototype statement. Thus, a function with an argument would be identified using the form:

```
function Function2Name( nArg )
```

Where further information is needed, the InstallShield IDE includes an extensive online help library with function definitions and explanations, sample code, and explanations of the various processes and structures that can be used.

Building a Disk Image

The final step (testing aside) in creating the application setup is to build a disk image using the Media Build Wizard.

The Media Build Wizard is used to create disk image folders on a local or network drive that contain all of the files needed for the application setup. A typical disk image folder includes one (or more) .CAB files that contain the actual application executable and associated files. In addition to the .CAB file, the folder will also include the Setup .exe program and other files used during installation. To create a distribution volume, the contents of the subdirectory are simply copied to a distributable media (or placed in a shared directory on a network).

NOTE

A setup program for the demo program `BatteryTest.exe` is found on the CD accompanying this book in the subdirectory `\Appendix B\Battery Test Install Disk`. The files in this directory can be copied to a 3.5" disk, or the `Setup.exe` program can be run directly from the CD.

To complete the task of creating a distributable application setup:

1. Run InstallShield, click the Media pane tab and then select the Media Build Wizard icon. Initially, the Media Build Wizard will ask for a media name. This can be any appropriate name such as "Battery Test Installation" or "Floppy Disk Install."
2. Media Build Wizard requests the media type. Options include 3.5" disks (from 1.44MB to 4.0MB), 5.25" disks (1.2MB), CD-ROM (650MB), and install from the Internet, as well as a custom size option.
3. After naming the media build and selecting the media type, Media Build Wizard will ask for the Build Type, either Full or Quick. The Quick option is used only for testing and the Full Build must be requested for distribution. Also in this step, the Review Report Before Build checkbox is selected by default. Clear the checkbox unless a preliminary report is desired.
4. While the Tag File dialog (Figure B.9) is used to identify each disk in a multi-disk distribution set, the information supplied here is optional.

FIGURE B.9:

The Tag File dialog

Media Build Wizard - Tag File

The following information will be stored in the tag file (.tag) placed on the built media:

Company Name: Reality Engineers

Application Name: BatteryTest

Version: 1.00.00.1

Product Category: System Tool

Misc:

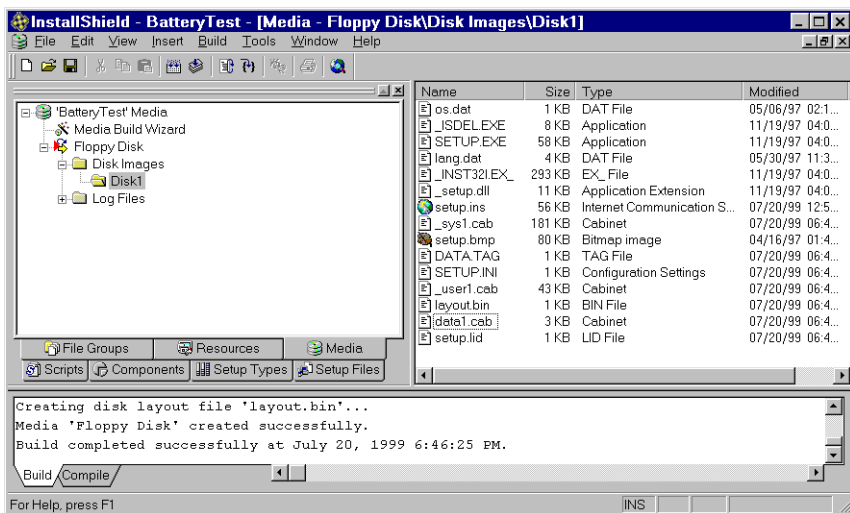
< Back Next > Cancel Help

- After confirming the supported platforms (operating systems and CPUs), the Summary panel will offer a review of the choices made. You may click the Back button repeatedly to step back through the Media Build Wizard screens or click Finish to build a media distribution set.

When the build is completed, the new build will appear on the Media tree, while the Media list will show the files included in the media build (Figure B.10).

FIGURE B.10:

The Completed Media Build



Limits and Shortcomings

While InstallShield for Windows does produce a working setup program (using the preceding instructions and steps), further refinements are also possible.

For one, the `setup.bmp` file used as a splash screen during setup is a default sample, and should ideally be replaced by a custom bitmap that is appropriate for your application.

Second, the license agreement is also a dummy and should be rewritten, as appropriate, to satisfy your legal and corporate requirements.

Third, the default setup, when executed, includes provisions for a serial number. While the default implementation will accept any entry as a valid serial number (or installation key), a more appropriate format would be to modify the setup code to either omit this element or to make provisions for actual verification.

Last, there are a variety of other features that can be used (or ignored) to customize the installation process as appropriate to your needs and requirements.

These brief caveats aside, one further curiosity, although not precisely a flaw, is found after the setup program is exercised. This discrepancy was noted in the Start menu where the installed application was correctly added, except that the icon provided was the default Visual C/C++ icon, not the application icon.

Why this happens is uncertain, since the application itself does display the correct icon and since the icon appearing on the menu is not found in the application. In any case, as this is a minor discrepancy, resolution and explanation are left to the interested reader.